Chapter 21

Incremental Redisplay

21.1 Overview of Incremental Redisplay

CLIM's incremental redisplay facility to allows the programmer to change the output in an output history (and hence, on the screen or other output device) in an incremental fashion. It allows the programmer to redisplay individual pieces of the existing output differently, under program control. It is "incremental" in the sense that CLIM will try to minimize the changes to the existing output on a display device when displaying new output.

There are two different ways to do incremental redisplay.

The first is to call `redisplay` on an output record. In essence, this tells CLIM to recompute the output of that output record over from scratch. CLIM compares the new results with the existing output and tries to do minimal redisplay. The `updating-output` form allows the programmer to assist CLIM by informing it that entire branches of the output history are known not to have changed. `updating-output` also allows the programmer to communicate the fact that a piece of the output record hierarchy has moved, either by having an output record change its parent, or by having an output record change its position.

The second way to do incremental redisplay is for the programmer to manually do the updates to the output history, and then call `note-output-record-child-changed` on an output record. This causes CLIM to propagate the changes up the output record tree and allows parent output records to readjust themselves to account for the changes.

Each style is appropriate under different circumstances. `redisplay` is often easier to use, especially when there might are large numbers of changes between two passes, or when the programmer has only a poor idea as to what the changes might be. `note-output-record-child-changed` can be more efficient for small changes at the bottom of the output record hierarchy, or in cases where the programmer is well informed as to the specific changes necessary and can help CLIM out.

21.1.1 Examples of Incremental Redisplay

The usual technique of incremental redisplay is to use `updating-output` to inform CLIM what output has changed, and use `redisplay` to recompute and redisplay that output.

The outermost call to `updating-output` identifies a program fragment that produces incrementally redisplayable output. A nested call to `updating-output` (that is, a call to `updating-output` that occurs during the execution of the body of the outermost `updating-output` and specifies the same stream) identifies an individually redisplayable piece of output, the program fragment that produces that output, and the circumstances under which that output needs to be redrawn. This nested calls to `updating-output` are just hints to incremental redisplay that can reduce the amount of work done by CLIM.

The outermost call to `updating-output` executes its body, producing the initial version of the output, and returns an `updating-output-record` that captures the body in a closure. Each nested call to `updating-output` stores its `:unique-id` and `:cache-value` arguments and the portion of the output produced by its body.

`redisplay` takes an `updating-output-record` and executes the captured body of `updating-output` over again. When a nested call to `updating-output` is executed during redisplay, `updating-output` decides whether the cached output can be reused or the output needs to be redrawn. This is controlled by the `:cache-value` argument to `updating-output`. If its value matches its previous value, the body would produce output identical to the previous output and thus it is unnecessary for CLIM to execute the body again. In this case the cached output is reused and `updating-output` does not execute its body. If the cache value does not match, the output needs to be recomputed, so `updating-output` executes its body and the new output drawn on the stream replaces the previous output. The `:cache-value` argument is only meaningful for nested calls to `updating-output`.

In order to compare the cache to the output record, two pieces of information are necessary:

- An association between the output being done by the program and a particular cache. This is supplied in the `:unique-id` option to `updating-output`.
- A means of determining whether this particular cache is valid. This is the `:cache-value` option to `updating-output`.

Normally, the programmer would supply both options. The unique-id would be some data structure associated with the corresponding part of output. The cache value would be something in that data structure that changes whenever the output changes.

It is valid to give the `:unique-id` and not the `:cache-value`. This is done to identify a parent in the hierarchy. By this means, the children essentially get a more complex unique id when they are matched for output. (In other words, it is like using a telephone area code.) The cache without a cache value is never valid. Its children always have to be checked.

It is also valid to give the `:cache-value` and not the `:unique-id`. In this case, unique ids are just assigned sequentially. So, if output associated with the same thing is done in the same order each time, it isn't necessary to invent new unique ids for each piece. This is especially true in the case of children of a cache with a unique id and no cache value of its own. In this case, the parent marks the particular data structure, whose components can change individually, and the children are always in the same order and properly identified by their parent and the order in which they are output.

A unique id need not be unique across the entire redisplay, only among the children of a given output cache; that is, among all possible (current and additional) uses made of `updating-output` that are dynamically (not lexically) within another.

To make incremental redisplay maximally efficient, the programmer should attempt to give as many caches with `:cache-value` as possible. For instance, if the thing being redisplayed is a deeply nested tree, it is better to be able to know when whole branches have not changed than to have to recurse to every single leaf and check it. So, if there is a modification tick in the leaves, it is better to also have one in their parent of the leaves and propagate the modification up when things change. While the simpler approach works, it requires CLIM to do more work than is necessary.

The following function illustrates the standard use of incremental redisplay:

```
(defun test (stream)
  (let* ((list (list 1 2 3 4 5))
         (record
           (updating-output (stream)
             (do* ((elements list (cdr elements))
                   (count 0 (1+ count)))
                  ((null elements))
               (let ((element (first elements)))
                 (updating-output (stream :unique-id count
                                          :cache-value element)
                   (format stream "Element ~D" element)
                   (terpri stream)))))))
    (sleep 10)
    (setf (nth 2 list) 17)
    (redisplay record stream)))
```

When this function is run on a window, the initial display will look like:

```
  Element 1
  Element 2
  Element 3
  Element 4
  Element 5
```

After the sleep has terminated, the display will look like:

```
  Element 1
  Element 2
  Element 17
  Element 4
  Element 5
```

CLIM takes care of ensuring that only the third line gets erased and redisplayed. In the case where items moved around (try the example substituting

```
(setq list (sort list #'(lambda (x y)
                          (declare (ignore x y))
                          (zerop (random 2)))))
```

for the form after the call to `sleep`), CLIM would ensure that the minimum amount of work would be done in updating the display, thereby minimizing "flashiness" while providing a powerful user interface.

See Chapter 28 for a discussion of how to use incremental redisplay automatically within the panes of an application frame.

21.2 Standard Programmer Interface

⇒ `updating-output` *(stream* `&rest` *args* `&key` *unique-id (id-test #'*`eql`*) cache-value (cache-test #'*`eql`*) fixed-position all-new parent-cache record-type)* `&body` *body* [*Macro*]

Introduces a caching point for incremental redisplay.

The *stream* argument is not evaluated, and must be a symbol that is bound to an output recording stream. If *stream* is `t`, `*standard-output*` is used. *body* may have zero or more declarations as its first forms.

record-type specifies the class of output record to create. The default is `standard-updating-output-record`. This argument should only be supplied by a programmer if there is a new class of output record that supports the updating output record protocol.

`updating-output` returns the output record it creates.

`updating-output` must be implemented by expanding into a call to `invoke-updating-output`, supplying a function that executes *body* as the *continuation* argument to `invoke-updating-output`. The exact behavior of this macro is described under `invoke-updating-output`.

⇒ `invoke-updating-output` *stream continuation record-type unique-id id-test cache-value cache-test* `&key` *all-new parent-cache* [*Generic Function*]

Introduces a caching point for incremental redisplay. Calls the function *continuation*, which generates the output records to be redisplayed. *continuation* is a function of one argument, the stream; it has dynamic extent.

If this is used outside the dynamic scope of an incremental redisplay, it has no particular effect. However, when incremental redisplay is occurring, the supplied *cache-value* is compared with the value stored in the cache identified by *unique-id*. If the values differ or the code in *body* has not been run before, the code in *body* runs, and *cache-value* is saved for next time. If the cache values are the same, the code in *body* is not run, because the current output is still valid.

unique-id provides a means to uniquely identify the output done by *body*. If *unique-id* is not supplied, CLIM will generate one that is guaranteed to be unique. *unique-id* may be any object as long as it is unique with respect to the *id-test* predicate among all such unique ids in the current incremental redisplay. *id-test* is a function of two arguments that is used for comparing unique ids; it has indefinite extent.

cache-value is a value that remains constant if and only if the output produced by body does not need to be recomputed. If the cache value is not supplied, CLIM will not use a cache for this piece of output. *cache-test* is a function of two arguments that is used for comparing cache values; it has indefinite extent.

If *fixed-position* is *true*, then the location of this output is fixed relative to its parent output record. When CLIM redisplays an output record that has a fixed position, then if the contents have not changed, the position of the output record will not change. If the contents have changed, CLIM assumes that the code will take care to preserve its position. The default for *fixed-position* is *false*.

If *all-new* is *true*, that indicates that all of the output done by *body* is new, and will never match output previously recorded. In this case, CLIM will discard the old output and do the redisplay from scratch. The default for *all-new* is *false*.

The output record tree created by `updating-output` defines a caching structure where mappings from a unique-id to an output record are maintained. If the programmer specifies an output record some output record P via the *parent-cache* argument, then CLIM will try to find a corresponding output record with the matching unique-id in the cache belonging to P. If neither *parent-cache* is not provided, then CLIM looks for the unique-id in the output record created by immediate dynamically enclosing call to `updating-output`. If that fails, CLIM use the unique-id to find an output record that is a child of the output history of *stream*. Once CLIM has found an output record that matches the unique-id, it uses the cache value and cache test to determine whether the output record has changed. If the output record has not changed, it may have moved, in which case CLIM will simply move the display of the output record on the display device.

`invoke-updating-output` returns the output record it creates.

⇒ `redisplay` *record stream* `&key` *(check-overlapping* `t`*)* [*Function*]

This function simply calls `redisplay-output-record` on the arguments *record* and *stream*.

⇒ `redisplay-output-record` *record stream* `&optional` *(check-overlapping* `t`*) x y parent-x parent-y* [*Generic Function*]

Minor issue: *The coordinate system stuff affected by the x/y and parent-x/y arguments is entirely bogus. The proposal to make "stream relative" coordinates for output records instead of "parent relative" coordinates will eliminate this completely. — SWM*

(`redisplay-output-record` *record stream*) causes the output of *record* to be recomputed. CLIM redisplays the changes "incrementally", that is, it only displays those parts that have been changed. *record* must already be part of the output history of the *output recording stream* *stream*, although it can be anywhere inside the hierarchy.

When *check-overlapping* is *false*, this means that CLIM can assume that no sibling output records overlap each other at any level in the output record tree. Supplying a *false* value for this argument can improve performance of redisplay.

Implementation note: `redisplay-output-record` is implemented by first binding `stream-redisplaying-p` of the stream to *true*, then creating the new output records by invoking `compute-new-output-records`. Once the new output records have been computed, `compute-difference-set` is called to compute the difference set, which is then passed to `note-child-output-record-changed`.

The other optional arguments can be used to specify where on the *stream* the output record should be redisplayed. *x* and *y* represent where the cursor should be, relative to (`output-record-parent` record), before we start redisplaying *record*. *parent-x* and *parent-y* can be supplied to say: do the output as if the parent started at positions *parent-x* and *parent-y* (which are in absolute coordinates). The default values for *x* and *y* are (`output-record-start-position` *record*). The default values for *parent-x* and *parent-y* are

```
(convert-from-relative-to-absolute-coordinates
  stream (output-record-parent record))
```

record will usually be an output record created by `updating-output`. If it is not, then `redisplay-output-record` will be equivalent to `replay-output-record`.

21.3 Incremental Redisplay Protocol

Major issue: *While the description of the API here is accurate, the description of the protocol is a disaster. This is no surprise, since the protocol for increment redisplay is itself a disaster. — SWM*

⇒ `updating-output-record` [*Protocol Class*]

The protocol class corresponding to records that support incremental redisplay; a subclass of `output-record`. If you want to create a new class that behaves like an updating output record, it should be a subclass of `updating-output-record`. All instantiable subclasses of `updating-output-record` must obey the updating output record protocol.

⇒ `updating-output-record-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is an *updating output record*, otherwise returns *false*.

⇒ `:unique-id` [*Initarg*]
⇒ `:id-test` [*Initarg*]
⇒ `:cache-value` [*Initarg*]
⇒ `:cache-test` [*Initarg*]
⇒ `:fixed-position` [*Initarg*]

All subclasses of `updating-output-record` must handle these four initargs, which are used to specify, respectively, the unique id and id test, cache value and cache test, and the "fixed position" component of the output record.

⇒ `standard-updating-output-record` [*Class*]

The instantiable class of output record that supports incremental redisplay. This is a subclass of `updating-output-record`.

⇒ `output-record-unique-id` *record* [*Generic Function*]

Returns the unique id associated with the updating output record *record*.

⇒ `output-record-cache-value` *record* [*Generic Function*]

Returns the cache value associated with the updating output record *record*.

⇒ `output-record-fixed-position` *record* [*Generic Function*]

Returns *true* if the updating output record *record* is at a fixed location on the output stream, otherwise returns *false*. Output records that are not at fixed location on the output stream will be moved by incremental redisplay when any of their siblings adjust their size or position.

⇒ `output-record-displayer` *record* [*Generic Function*]

Returns the function that produces the output for this output record. This is the function that is called during redisplay to produce new output if the cache value mismatches.

⇒ `compute-new-output-records` *record stream* [*Generic Function*]

`compute-new-output-records` modifies an output record tree to reflect new output done by the application. In addition to inserting the new output records into the output record tree, it must save enough information to be able to compute the difference set, such as the old bounding rectangle, old cursor positions, old children, and so forth.

`compute-new-output-records` recursively invokes itself on each child of *record*.

`compute-new-output-records` of an output record of type `updating-output-record` runs the displayer (`output-record-displayer`), which gives the behavior of incremental redisplay. That is, it reruns the code (getting hints from `updating-output`) and figures out the changes from there by comparing it to the old output history.

⇒ `compute-difference-set` *record* `&optional` *(check-overlapping* `t`*) (offset-x* `0`*) (offset-y* `0`*) (old-offset-x* `0`*) (old-offset-y* `0`*)* [*Generic Function*]

`compute-difference-set` compares the current state of the *output record record* with its previous state, and returns a "difference set" as five values. The difference set controls what needs to be done to the display device in order to accomplish the incremental redisplay.

The values returned are *erases* (what areas of the display device need to be erased), *moves* (what output records need to be moved), *draws* (what output records need to be freshly replayed), *erase-overlapping*, and *move-overlapping*. Each is a list whose elements are lists of the form:

When *check-overlapping* is *false*, this means that CLIM can assume that no sibling output records overlap each other at any level. Supplying a *false* value for this argument can improve performance of redisplay.

- *erases* are lists of (*record old–box*)
- *moves* are lists of (*record old–box new–position*)
- *draws* are lists of (*record old–box*)
- *erase-overlapping* is a list of (*record old–box*)
- *move-overlapping* is a list of (*record old–box new–position*)

⇒ `augment-draw-set` *record erases moves draws erase-overlapping move-overlapping* `&optional` *x-offset y-offset old-x-offset old-y-offset* [*Generic Function*]

Minor issue: *To be supplied. — SWM*

⇒ `note-output-record-child-changed` *record child mode old-position old-bounding-rectangle stream* `&optional` *erases moves draws erase-overlapping move-overlapping* `&key` *check-overlapping* [*Generic Function*]

`note-output-record-child-changed` is called after an output history has had changes made to it, but before any of the new output has been displayed. It will call `propagate-output-record-changes-p` to determine if the parent output record should be notified, and if so, will call `propagate-output-record-changes` to create an updated difference set. If no changes need to be propagated to the parent output record, then `note-output-record-child-changed` will call `incremental-redisplay` in order display the difference set.

mode is one of `:delete`, `:add`, `:change`, `:move`, or `:none`

old-position and *old-bounding-rectangle* describe where *child* was before it was moved.

check-overlapping is as for `compute-difference-set`.

⇒ `propagate-output-record-changes-p` *record child mode old-position old-bounding-rectangle* [*Generic Function*] `propagate-output-record-changes-p` is a predicate that returns *true* if the change made to the child will cause *record* to be redisplayed in any way. Otherwise, it returns *false*. *mode* is one of `:delete`, `:add`, `:change`, `:move`, or `:none`.

⇒ `propagate-output-record-changes` *record child mode* `&optional` *old-position old-bounding-rectangle erases moves draws erase-overlapping move-overlapping check-overlapping* [*Generic Function*]

Called when the changed *child* output record requires that its parent, *record*, be redisplayed as well. `propagate-output-record-changes` will update the difference set to reflect the additional changes.

check-overlapping is as for `compute-difference-set`.

⇒ `match-output-records` *record* `&rest` *initargs* [*Generic Function*]

Returns *true* if record matches the supplied class initargs *initargs*, otherwise returns *false*.

⇒ `find-child-output-record` *record use-old-elements record-type* `&rest` *initargs* `&key` *unique-id unique-id-test* [*Generic Function*]

Finds a child of *record* matching the *record-type* and the supplied initargs *initargs*. *unique-id* and *unique-id-test* are used to match against the children as well. *use-old-elements* controls whether the desired record is to be found in the previous (before redisplay) contents of the record.

⇒ `output-record-contents-ok` *record* [*Generic Function*]

Returns *true* if the current state of *record* are up to date, otherwise returns *false*.

⇒ `recompute-contents-ok` *record* [*Generic Function*]

Compares the old (before redisplay) and new contents of *record* to determine whether or not this record changed in such a way so that the display needs updating.

⇒ `cache-output-record` *record child unique-id* [*Generic Function*]

record stores *child* such that it can be located later using *unique-id*.

⇒ `decache-child-output-record` *record child use-old-elements* [*Generic Function*]

Invalidates the redisplay state of *record*.

⇒ `find-cached-output-record` *record use-old-elements record-type* `&rest` *initargs* `&key` *unique-id unique-id-test* `&allow-other-keys` [*Generic Function*]

Finds a previously cached child matching *record-type*, *initargs*, *unique-id*, and *unique-id-test*. *use-old-elements* controls whether the desired record is to be found in the previous (before redisplay) contents of the record.

21.4 Incremental Redisplay Stream Protocol

⇒ `redisplayable-stream-p` *stream* [*Generic Function*]

Returns *true* for any stream that maintains an output history and supports the incremental redisplay protocol, otherwise returns *false*.

⇒ `stream-redisplaying-p` *stream* [*Generic Function*]

Returns *true* if the *stream* is currently doing redisplay (that is, is inside of a call to `redisplay`), otherwise returns *false*.

⇒ `incremental-redisplay` *stream position erases moves draws erase-overlapping move-overlapping* [*Generic Function*]

Performs the incremental update on *stream* according to the difference set comprised by *erases*, *moves*, *draws*, *erase-overlapping*, and *move-overlapping*, which are values returned by `compute-difference-set`. *position* is a point object that represents the start position of the topmost output record that will be redisplayed.

`incremental-redisplay` can be called on any extended output stream.

Part VI

Extended Stream Input Facilities

Chapter 22

Extended Stream Input

CLIM provides a stream-oriented input layer that is implemented on top of the sheet input architecture. The basic CLIM input stream protocol is based on the character input stream protocol proposal submitted to the ANSI Common Lisp committee by David Gray. This proposal was not approved by the committee, but has been implemented by most Lisp vendors.

22.1 Basic Input Streams

CLIM provides an implementation of the basic input stream facilities (described in more detail in Appendix D), either by directly using the underlying Lisp implementation, or by implementing the facilities itself.

⇒ `standard-input-stream` [*Class*]

This class provides an implementation of the CLIM's basic input stream protocol based on CLIM's input kernel. It defines a `handle-event` method for keystroke events and queues the resulting characters in a per-stream input buffer. Members of this class are mutable.

⇒ `stream-read-char` *stream* [*Generic Function*]

Returns the next character available in the *input stream stream*, or `:eof` if the stream is at end-of-file. If no character is available this function will wait until one becomes available.

⇒ `stream-read-char-no-hang` *stream* [*Generic Function*]

Like `stream-read-char`, except that if no character is available the function returns *false*.

⇒ `stream-unread-char` *stream character* [*Generic Function*]

Places the character *character* back into the *input stream stream*'s input buffer. The next call to `read-char` on *stream* will return the unread character. The character supplied must be the most recent character read from the stream.

⇒ `stream-peek-char` *stream* [*Generic Function*]

Returns the next character available in the *input stream stream*. The character is not removed from the input buffer. Thus, the same character will be returned by a subsequent call to `stream-read-char`.

⇒ `stream-listen` *stream* [*Generic Function*]

Returns *true* if there is input available on the *input stream stream*, *false* if not.

⇒ `stream-read-line` *stream* [*Generic Function*]

Reads and returns a string containing a line of text from the *input stream stream*, delimited by the `#\Newline` character.

⇒ `stream-clear-input` *stream* [*Generic Function*]

Clears any buffered input associated with the *input stream stream*, and returns *false*.

22.2 Extended Input Streams

In addition to the basic input stream protocol, CLIM defines an extended input stream protocol. This protocol extends the stream model to allow manipulation of non-character user gestures, such as pointer button presses. The extended input protocol provides the programmer with more control over input processing, including the options of specifying input wait timeouts and auxiliary input test functions.

⇒ `extended-input-stream` [*Protocol Class*]

The protocol class for CLIM extended input streams. This is a subclass of `input-stream`. If you want to create a new class that behaves like an extended input stream, it should be a subclass of `extended-input-stream`. All instantiable subclasses of `extended-input-stream` must obey the extended input stream protocol.

⇒ `extended-input-stream-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is a CLIM *extended input stream*, otherwise returns *false*.

⇒ `:input-buffer` [*Initarg*]
⇒ `:pointer` [*Initarg*]
⇒ `:text-cursor` [*Initarg*]

All subclasses of `extended-input-stream` must handle these initargs, which are used to specify, respectively, the input buffer, pointer, and text cursor for the extended input stream.

⇒ `standard-extended-input-stream` [*Class*]

This class provides an implementation of the CLIM extended input stream protocol based on CLIM's input kernel. The extended input stream maintains the state of the display's pointing devices (such as a mouse) in pointer objects associated with the stream. It defines a **`handle-event`** methods for keystroke and pointer motion and button press events and updates the pointer object state and queues the resulting events in a per-stream input buffer.

Members of this class are mutable.

22.2.1 The Extended Input Stream Protocol

The following generic functions comprise the extended input stream protocol. All extended input streams must implement methods for these generic functions.

⇒ `stream-input-buffer` *stream* [*Generic Function*]

⇒ `(setf stream-input-buffer)` *buffer stream* [*Generic Function*]

The functions provide access to the stream's input buffer. Normally programs do not need to manipulate the input buffer directly. It is sometimes useful to cause several streams to share the same input buffer so that input that comes in on one of them is available to an input call on any of the streams. The input buffer must be vector with a fill pointer capable of holding general input gesture objects (such as characters and event objects).

⇒ `stream-pointer-position` *stream* `&key` *pointer* [*Generic Function*]

Returns the current position of the pointing device *pointer* for the *extended input stream stream* as two values, the x and y positions in the stream's drawing surface coordinate system. If *pointer* is not supplied, it defaults to `port-pointer` of the stream's port.

⇒ `(setf* stream-pointer-position)` *x y stream* `&key` *pointer* [*Generic Function*]

Sets the position of the pointing device for the *extended input stream stream* to x and y, which are integers. *pointer* is as for `stream-pointer-position`.

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

⇒ `stream-set-input-focus` *stream* [*Generic Function*]

Sets the "input focus" to the *extended input stream stream* by changing the value of `port-keyboard-input-focus` and returns the old input focus as its value.

⇒ `with-input-focus` *(stream)* `&body` *body* [*Macro*]

Temporarily gives the keyboard input focus to the *extended input stream stream*. By default, an application frame gives the input focus to the window associated with `frame-query-io`.

The *stream* argument is not evaluated, and must be a symbol that is bound to a stream. If *stream* is `t`, `*standard-input*` is used. *body* may have zero or more declarations as its first forms.

⇒ `*input-wait-test*` [*Variable*]
⇒ `*input-wait-handler*` [*Variable*]
⇒ `*pointer-button-press-handler*` [*Variable*]

These three variables are used to hold the default values for the current input wait test, wait handler, and pointer button press handler. These variables are globally bound to `nil`.

⇒ `read-gesture &key` *(stream* `*standard-input*`*) timeout peek-p (input-wait-test* `*input-wait-test*`*) (input-wait-handler* `*input-wait-handler*`*) (pointer-button-press-handler* `*pointer-button-press-handler*`*)* [*Function*]

Calls `stream-read-gesture` on the *extended input stream stream* and all of the other keyword arguments. These arguments are the same as for `stream-read-gesture`.

⇒ `stream-read-gesture` *stream* `&key` *timeout peek-p (input-wait-test* `*input-wait-test*`*) (input-wait-handler* `*input-wait-handler*`*) (pointer-button-press-handler* `*pointer-button-press-handler*`*)* [*Generic Function*]

Returns the next gesture available in the *extended input stream stream*; the gesture will be either a character or an event (such as a pointer button event). The input is not echoed.

If the user types an abort gesture (that is, a gesture that matches any of the gesture names in `*abort-gestures*`), then the `abort-gesture` condition will be signalled.

If the user types an accelerator gesture (that is, a gesture that matches any of the gesture names in `*accelerator-gestures*`), then the `accelerator-gesture` condition will be signalled.

`stream-read-gesture` works by invoking `stream-input-wait` on *stream*, *input-wait-test*, and *timeout*, and then processing the input, if there is any. `:around` methods on this generic function can be used to implement some sort of a gesture preprocessing mechanism on every gesture; CLIM's input editor will typically be implemented this way.

timeout is either `nil` or an integer that specifies the number of seconds that `stream-read-gesture` will wait for input to become available. If no input is available, `stream-read-gesture` will return two values, `nil` and `:timeout`.

If the boolean *peek-p* is *true*, then the returned gesture will be left in the stream's input buffer.

input-wait-test is a function of one argument, the stream. The function should return *true* when there is input to process, otherwise it should return *false*. This argument will be passed on to `stream-input-wait`. `stream-read-gesture` will bind `*input-wait-test*` to *input-wait-test*.

input-wait-handler is a function of one argument, the stream. It is called when `stream-input-wait` returns *false* (that is, no input is available). This option can be used in conjunction with *input-wait-test* to handle conditions other than keyboard gestures, or to provide some sort of interactive behavior (such as highlighting applicable presentations). `stream-read-gesture` will bind `*input-wait-handler*` to *input-wait-handler*.

pointer-button-press-handler is a function of two arguments, the stream and a pointer button press event. It is called when the user clicks a pointer button. `stream-read-gesture` will bind `*pointer-button-press-handler*` to *pointer-button-press-handler*.

input-wait-test, *input-wait-handler*, and *pointer-button-press-handler* have dynamic extent.

⇒ `stream-input-wait` *stream* `&key` *timeout input-wait-test* [*Generic Function*]

Waits for input to become available on the *extended input stream stream*. *timeout* and *input-wait-test* are as for `stream-read-gesture`.

⇒ `unread-gesture` *gesture* `&key` *(stream* `*standard-input*`*)* [*Function*]

Calls `stream-unread-gesture` on *gesture* and *stream*. These arguments are the same as for `stream-unread-gesture`.

⇒ `stream-unread-gesture` *stream gesture* [*Generic Function*]

Places *gesture* back into the *extended input stream stream*'s input buffer. The next call to `stream-read-gesture` request will return the unread gesture. The gesture supplied must be the most recent gesture read from the stream via `read-gesture`.

22.2.2 Extended Input Stream Conditions

⇒ `*abort-gestures*` [*Variable*]

A list of all of the gesture names that correspond to abort gestures. The exact global set of standard abort gestures is unspecified, but must include the `:abort` gesture name.

⇒ `abort-gesture` [*Condition*]

This condition is signalled by `read-gesture` whenever an abort gesture (one of the gestures in `*abort-gestures*` is read from the user. This condition will handle the `:event` initarg, which is used to supply the event corresponding to the abort gesture.

⇒ `abort-gesture-event` *condition* [*Generic Function*]

Returns the event that cause the abort gesture condition to be signalled. `condition` is an object of type `abort-gesture`.

⇒ `*accelerator-gestures*` [*Variable*]

A list of all of the gesture names that correspond to keystroke accelerators. The global value for this is `nil`.

⇒ `accelerator-gesture` [*Condition*]

This condition is signalled by `read-gesture` whenever an keystroke accelerator gesture (one of the gestures in `*accelerator-gestures*` is read from the user. This condition will handle the

`:event` and the `:numeric-argument` initargs, which are used to supply the event corresponding to the abort gesture and the accumulated numeric argument (which defaults to 1).

⇒ `accelerator-gesture-event` *condition* [*Generic Function*]

Returns the event that caused the accelerator gesture condition to be signalled. `condition` is an object of type `accelerator-gesture`.

⇒ `accelerator-gesture-numeric-argument` *condition* [*Generic Function*]

Returns the accumlated numeric argument (maintained by the input editor) at the time the accelerator gesture condition was signalled. `condition` is an object of type `accelerator-gesture`.

22.3 Gestures and Gesture Names

A *gesture* is some sort of input action by the user, such as typing a character or clicking a pointer button. A *keyboard gesture* refers to those gestures that are input by typing something on the keyboard. A *pointer gesture* refers to those gestures that are input by doing something with the pointer, such as clicking a button.

A *gesture name* is a symbol that gives a name to a set of similar gestures. Gesture names are used in order to provide a level of abstraction above raw device events; greater portability can thus be achieved by avoiding referring directly to platform-dependent constructs, such as character objects that refer to a particular key on the keyboard. For example, the `:complete` gesture is used to name the gesture that causes the `complete-input` complete the current input string; on Genera, this may correspond to the Complete key on the keyboard (which generates a `#\Complete` character), but on a Unix workstation, it may correspond to some other key. Another example is `:select`, which is commonly used to indicate a left button click on the pointer.

Note that gesture names participate in a one-to-many mapping, that is, a single gesture name can name a group of physical gestures. For example, an `:edit` might include both a pointer button click and a key press.

CLIM uses *event* objects to represent user gestures. Some of the more common events are those of the class `pointer-button-event`. Event objects store the sheet associated with the event, a timestamp, and the modifier key state (a quantity that indicates which modifier keys were held down on the keyboard at the time the event occurred). Pointer button event objects also store the pointer object, the button that was clicked on the pointer, the window the pointer was over and the x and y position within that window. Keyboard gestures store the key name.

In some contexts, the object used to represent a user gesture is referred to as an *gesture object*. An gesture object might be exactly the same as an event object, or might contain less information. For example, for a keyboard gesture that corresponds to a standard printing character, it may be enough to represent the gesture object as a character.

⇒ `define-gesture-name` *name type gesture-spec* `&key` *(unique* `t`*)* [*Macro*]

Defines a new gesture named by the symbol *name*. *type* is the type of gesture being created, and must be one of the symbols described below. *gesture-spec* specifies the physical gesture that corresponds to the named gesture; its syntax depends on the value of *type*. `define-gesture-name` must expand into a call to `add-gesture-name`.

If *unique* is *true*, all old gestures named by *name* are first removed. *unique* defaults to `t`.

None of the arguments to `define-gesture-name` is evaluated.

⇒ `add-gesture-name` *name type gesture-spec* `&key` *unique* [*Function*]

Adds a gesture named by the symbol *name* to the set of gesture names. *type* is the type of gesture being created, and must be one of the symbols described below. *gesture-spec* specifies the physical gesture that corresponds to the named gesture; its syntax depends on the value of *type*.

If *unique* is *true*, all old gestures named by *name* are first removed. *unique* defaults to `nil`.

When *type* is `:keyboard`, *gesture-spec* is a list of the form *(key-name . modifier-key-names)*. *key-name* is the name of a non-modifier key on the keyboard (see below). *modifier-key-names* is a (possibly empty) list of modifier key names (`:shift`, `:control`, `:meta`, `:super`, and `:hyper`).

For the standard Common Lisp characters (the 95 ASCII printing characters including `#\Space`), *key-name* is the character object itself. For the other "semi-standard" characters, *key-name* is a keyword symbol naming the character (`:newline`, `:linefeed`, `:return`, `:tab`, `:backspace`, `:page`, and `:rubout`). CLIM implementations may extend the set of key names on a per-port basic, but should choose a port-specific package. For example, the Genera port might such gestures as include `genera-clim:help` and `genera-clim:complete`.

The names of the modifier keys have been chosen to be uniform across all platforms, even though not all platforms will have keys on the keyboard with these names. The per-port part of a CLIM implementation must simply choose a sensible mapping from the modifier key names to the names of the keys on the keyboard. For example, a CLIM implementation on the Macintosh might map `:meta` to the Command shift key, and `:super` to the Option shift key.

When *type* is `:pointer-button`, `:pointer-button-press`, or `:pointer-button-release`, *gesture-spec* is a list of the form *(button-name . modifier-key-names)*. *button* is the name of a pointer button (`:left`, `:middle`, or `:right`), and *modifier-key-names* is as above.

CLIM implementations are permitted to have other values of *type* as an extension, such as `:pointer-motion` or `:timer`.

As an example, the `:edit` gesture name above could be defined as follows using `define-gesture-name`:

```
(define-gesture-name :edit :pointer-button (:left :meta))
(define-gesture-name :edit :keyboard (#\E :control))
```

⇒ `delete-gesture-name` *name* [*Function*]

Removes the gesture named by the symbol *name*.

⇒ `event-matches-gesture-name-p` *event gesture-name* [*Function*]

Returns *true* if the device event *event* "matches" the gesture named by *gesture-name*.

For pointer button events, the event matches the gesture name when the pointer button from the event matches the name of the pointer button one of the gesture specifications named by *gesture-name*, and the modifier key state from the event matches the names of the modifier keys in that same gesture specification.

For keyboard events, the event matches the gesture name when the key name from the event matches the key name of one of the gesture specifications named by *gesture-name*, and the modifier key state from the event matches the names of the modifier keys in that same gesture specification.

⇒ `modifier-state-matches-gesture-name-p` *modifier-state gesture-name* [*Function*]

Returns *true* if the modifier key state from the device event *event* matches the names of the modifier keys in one of the gesture specifications named by *gesture-name*.

Minor issue: *Note that none of the functions above take a port argument. This is because CLIM implicitly assumes that the canonical set of gesture names is the same on every port, and only the mappings differ from port to port. Some ports may define additional gesture names, but they will simply not be mapped on other ports. Is this a reasonable assumption? — SWM*

⇒ `make-modifier-state &rest` *modifiers* [*Function*]

Given a list of modifier state names, this creates an integer that serves as a modifier key state. The legal modifier state names are `:shift`, `:control`, `:meta`, `:super`, and `:hyper`.

22.3.1 Standard Gesture Names

Every CLIM implementation must provide a standard set of gesture names that correspond to a common set of gestures. These gesture names must have a meaningful mapping for every port type.

Here are the required, standard keyboard gesture names:

- `:abort`—corresponds to gestures that cause the currently running application to be aborted back to top-level. On Genera, this will match the `#\Abort` character. On other systems, this may match the event corresponding to typing `Control-C`.
- `:clear-input`—corresponds to gestures that cause the current input buffer to be cleared. On Genera, this will match the `#\Clear-Input` character. On other systems, this may match the event corresponding to typing `Control-U`.
- `:complete`—corresponds to the gestures that tell the completion facility to complete the

current input. On most systems, this will typically match the `#\Tab` or `#\Escape` character. On Genera, this will match the `#\Complete` character as well.

- `:help`—corresponds to the gestures that tell `accept` and the completion facility to display a help message. On most systems, this will typically match the event corresponding to typing `Control-/`. On Genera, this will match the `#\Help` character as well.
- `:possibilities`—corresponds to the gestures that tell the completion facility to display the current set of possible completions. On most systems, this will typically match the event corresponding to typing `Control-?`.

Here are the required, standard pointer gesture names:

- `:select`—corresponds to the gesture that is used to "select" the object being pointed to with the pointer. Typically, this will correspond to the left button on the pointer.
- `:describe`—corresponds to the gesture that is used to "describe" or display some sort of documentation on the object being pointed to with the pointer. Typically, this will correspond to the middle button on the pointer.
- `:menu`—corresponds to the gesture that is used to display a menu of all possible operation on the object being pointed to with the pointer. Typically, this will correspond to the right button on the pointer.
- `:edit`—corresponds to the gesture that is used to "edit" the object being pointed to with the pointer. Typically, this will correspond to the left button on the pointer with some modifier key held down (such as the `:meta` key).
- `:delete`—corresponds to the gesture that is used to "delete" the object being pointed to with the pointer. Typically, this will correspond to the middle button on the pointer with some modifier key held down (such as the `:shift` key).

22.4 The Pointer Protocol

⇒ `pointer` *[Protocol Class]*

The protocol class that corresponds to a pointing device. If you want to create a new class that behaves like a pointer, it should be a subclass of `pointer`. All instantiable subclasses of `pointer` must obey the pointer protocol. Members of this class are mutable.

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

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

⇒ `:port` *[Initarg]*

The `:port` initarg is used to specify the port with which the pointer is associated.

⇒ `standard-pointer` *[Class]*

The instantiable class that implements a pointer.

⇒ `pointer-sheet` *pointer* [*Generic Function*]
⇒ `(setf pointer-sheet)` *sheet pointer* [*Generic Function*]

Returns (or sets) the sheet over which the pointer *pointer* is located.

⇒ `pointer-button-state` *pointer* [*Generic Function*]

Returns the current state of the buttons of the pointer *pointer* as an integer. This will be a mask consisting of the `logior` of `+pointer-left-button+`, `+pointer-middle-button+`, and `+pointer-right-button+`.

⇒ `pointer-position` *pointer* [*Generic Function*]

Returns the x and y position of the pointer *pointer* as two values. x and y are in the coordinates of the pointer's sheet.

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

Sets the x and y position of the pointer *pointer* to the specified position. x and y are in the coordinates of the pointer's sheet.

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

⇒ `pointer-native-position` *pointer* [*Generic Function*]

Returns the "native" x and y position of the pointer *pointer* as two values. x and y are in "graft" coordinates.

⇒ `(setf* pointer-native-position)` *x y pointer* [*Generic Function*]

Sets the native x and y position of the pointer *pointer* to the specified position. x and y are in "graft" coordinates.

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

⇒ `pointer-cursor` *pointer* [*Generic Function*]
⇒ `(setf pointer-cursor)` *cursor pointer* [*Generic Function*]

A pointer object usually has a visible cursor associated with it. These functions return (or set) the cursor associated with the pointer *pointer*.

⇒ `port` *(pointer* `standard-pointer`*)* [*Method*]

Returns the port with which *pointer* is associated.

22.5 Pointer Tracking

⇒ `tracking-pointer` *(sheet* `&key` *pointer multiple-window transformp context-type highlight)* `&body` *body* [*Macro*]

The `tracking-pointer` macro provides a general means for running code while following the position of a pointing device, and monitoring for other input events. The programmer supplies code (the clauses in *body*) to be run upon the occurrence of any of the following types of events:

- Motion of the pointer
- Motion of the pointer over a presentation
- Clicking or releasing a pointer button
- Clicking or releasing a pointer button while the pointer is over a presentation
- Keyboard event (typing a character)

The *sheet* argument is not evaluated, and must be a symbol that is bound to an input sheet or stream. If *sheet* is `t`, `*standard-output*` is used. *body* may have zero or more declarations as its first forms.

The *pointer* argument specifies a pointer to track. It defaults to the primary pointer for the sheet, `(port-pointer (port` *sheet*`))`.

When the boolean *multiple-windows* is *true*, then the pointer will be tracked across multiple windows, otherwise is will be tracked only in the window corresponding to *sheet*.

When the boolean *transformp* is *true*, then the coordinates supplied to the `:pointer-motion` clause will be in the "user" coordinate system rather than in stream coordinates, that is, the medium's transformation will be applied to the coordinates.

context-type is used to specify the presentation type of presentations that will be "visible" to the tracking code for purposes of highlighting and for the `:presentation`, `:presentation-button-press`, and `:presentation-button-release` clauses. Supplying *context-type* is only useful when *sheet* is an output recording stream. *context-type* defaults to `t`, meaning that all presentations are visible.

When *highlight* is *true*, `tracking-pointer` will highlight applicable presentations as the pointer is positioned over them. highlight defaults to *true* when any of the `:presentation`, `:presentation-button-press`, or `:presentation-button-release` clauses is supplied, otherwise it defaults to *false*. See Chapter 16 for a complete discussion of presentations.

The body of `tracking-pointer` consists of a list of clauses. Each clause is of the form
(clause-keyword arglist . clause-body)
and defines a local function to be run upon occurrence of each type of event. The possible values for *clause-keyword* and the associated *arglist* are:

- `:pointer-motion` *(*`&key` *window x y)*
  Defines a clause to run whenever the pointer moves. In the clause, *window* is bound to the window in which the motion occurred, and *x* and *y* to the coordinates of the pointer. (See the keyword argument `:transformp` below for a description of the coordinate system in which *x* and *y* are expressed.)

- `:presentation` *(*`&key` *presentation window x y)*
  Defines a clause to run whenever the pointer moves over a presentation of the desired type. (See the keyword argument `:context-type` above for a description of how to specify the desired type.) In the clause, *presentation* is bound to the presentation, *window* to the window in which the motion occurred, and *x* and *y* to the coordinates of the pointer. (See the keyword argument `:transformp` above for a description of the coordinate system in which *x* and *y* are expressed.)

  When both `:presentation` and `:pointer-motion` clauses are provided, the two clauses are mutually exclusive. The `:presentation` clause will run only if the pointer is over an applicable presentation, otherwise the `:pointer-motion` clause will run.

- `:pointer-button-press` *(*`&key` *event x y)*
  Defines a clause to run whenever a pointer button is pressed. In the clause, *event* is bound to the pointer button press event. (The window and the coordinates of the pointer are part of *event*.)

  x and *y* are the transformed *x* and *y* positions of the pointer. These will be different from `pointer-event-x` and `pointer-event-y` if the user transformation is not the identity transformation.

- `:presentation-button-press` *(*`&key` *presentation event x y)*
  Defines a clause to run whenever the pointer button is pressed while the pointer is over a presentation of the desired type. (See the keyword argument `:context-type` below for a description of how to specify the desired type.) In the clause, *presentation* is bound to the presentation, and *event* to the pointer button press event. (The window and the stream coordinates of the pointer are part of *event*.) *x* and *y* are as for the `:pointer-button-press` clause.

  When both `:presentation-button-press` and `:pointer-button-press` clauses are provided, the two clauses are mutually exclusive. The `:presentation-button-press` clause will run only if the pointer is over an applicable presentation, otherwise the `:pointer-button-press` clause will run.

- `:pointer-button-release` *(*`&key` *event x y)*
  Defines a clause to run whenever a pointer button is released. In the clause, *event* is bound to the pointer button release event. (The window and the coordinates of the pointer are part of *event*.)

  x and *y* are the transformed *x* and *y* positions of the pointer. These will be different from `pointer-event-x` and `pointer-event-y` if the user transformation is not the identity transformation.

- `:presentation-button-release` *(*`&key` *presentation event x y)*
  Defines a clause to run whenever a pointer button is released while the pointer is over a presentation of the desired type. (See the keyword argument `:context-type` below for a description of how to specify the desired type.) In the clause, *presentation* is bound to the presentation, and *event* to the pointer button release event. (The window and the stream

coordinates of the pointer are part of *event*.) *x* and *y* are as for the `:pointer-button-release` clause.

When both `:presentation-button-release` and `:pointer-button-release` clauses are provided, the two clauses are mutually exclusive. The `:presentation-button-release` clause will run only if the pointer is over an applicable presentation, otherwise the `:pointer-button-release` clause will run.

- `:keyboard` *(*`&key` *gesture)*
  Defines a clause to run whenever a character is typed on the keyboard. In the clause, *gesture* is bound to the keyboard gesture corresponding to the character typed.

⇒ `drag-output-record` *stream output-record* `&key` *repaint erase feedback finish-on-release multiple-window* [*Generic Function*]

Enters an interaction mode in which the user moves the pointer and *output-record* "follows" the pointer by being dragged on the *output recording stream stream*. By default, the dragging is accomplished by erasing the output record from its previous position and redrawing at the new position. *output-record* remains in the output history of *stream* at its final position.

The returned values are the final *x* and *y* position of the pointer.

The boolean *repaint* allows the programmer to control the appearance of windows as the pointer is dragged. If *repaint* is *true* (the default), displayed contents of windows are not disturbed as the output record is dragged over them (that is, those regions of the screen are repainted). If it is *false*, then no repainting is done as the output record is dragged.

erase allows the programmer to identify a function that will be called to erase the output record as it is dragged. It must be a function of two arguments, the output record to erase and the stream; it has dynamic extent. The default is `erase-output-record`.

feedback allows the programmer to identify a "feedback" function. *feedback* must be a is a function of seven arguments: the output record, the stream, the initial *x* and *y* position of the pointer, the current *x* and *y* position of the pointer, and a drawing argument (either `:erase` or `:draw`). It has dynamic extent. The default is `nil`, meaning that the feedback behavior will be for the output record to track the pointer. (The *feedback* argument is used when the programmer desires more complex feedback behavior, such as drawing a "rubber band" line as the user moves the mouse.) Note that if *feedback* is supplied, *erase* is ignored.

If the boolean *finish-on-release* is *false* (the default), `drag-output-record` is exited when the user presses a pointer button. When it is *true*, `drag-output-record` is exited when the user releases the pointer button currently being held down.

multiple-window is as for `tracking-pointer`.

⇒ `dragging-output` *(*`&optional` *stream* `&key` *repaint finish-on-release multiple-window)* `&body` *body* [*Macro*]

Evaluates *body* inside of `with-output-to-output-record` to produce an output record for the stream *stream*, and then invokes `drag-output-record` on the record in order to drag the output. The output record is not inserted into *stream*'s output history.

The returned values are the final x and y position of the pointer.

The *stream* argument is not evaluated, and must be a symbol that is bound to an *output recording stream* stream. If *stream* is `t` (the default), `*standard-output*` is used. *body* may have zero or more declarations as its first forms.

repaint, *finish-on-release*, and *multiple-window* are as for `drag-output-record`.

Chapter 23

Presentation Types

23.1 Overview of Presentation Types

The core around which the CLIM application user interface model is built is the concept of the application-defined user interface data type. Each application has its own set of semantically significant user interface entities; a CAD program for designing circuits has its various kinds of components (gates, resistors, and so on), while a database manager has its relations and field types. These entities have to be displayed to the user (possibly in more than one displayed representation) and the user has to be able to interact with and specify the entities via pointer gestures and keyboard input. Frequently each user interface entity has a corresponding Lisp data type (such as an application-specific structure or CLOS class definition), but this is not always the case. The data representation for an interaction entity may be a primitive Lisp data type. In fact, it is possible for several different user interface entities to use the same Lisp data type for their internal representation, for example, building floor numbers and employee vacation day totals could both be represented internally as integers.

CLIM provides a framework for defining the appearance and behavior of these user interface entities via the *presentation type* mechanism. A presentation type can be thought of as a CLOS class that has some additional functionality pertaining to its roles in the user interface of an application. By defining a presentation type the application programmer defines all of the user interface components of the entity:

- Its displayed representation, textual or graphical
- Textual representation, for user input via the keyboard
- Pointer sensitivity, for user input via the pointer

In other words, by defining a presentation type, the application programmer describes in one place all the information about an object necessary to display it to the user and interact with the user for object input.

The set of presentation types forms a type lattice, an extension of the Common Lisp CLOS type lattice. When a new presentation type is defined as a subtype of another presentation type it inherits all the attributes of the supertype except those explicitly overridden in the definition.

Minor issue: *Describe what a presentation type is more exactly. What is a parameterized presentation type? Why do we want them? Why are they in a lattice? How do they relate to CL types and CLOS classes? What exactly gets inherited? — SWM*

23.2 Presentations

A *presentation* is a special kind of output record that remembers not only output, but the object associated with the output and the semantic type associated with that object.

Minor issue: *Describe exactly what a presentation is. What does it mean for presentations to be nested? — SWM*

⇒ `presentation` [*Protocol Class*]

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

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

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

⇒ `standard-presentation` [*Class*]

The instantiable output record class that represents presentations. `present` normally creates output records of this class. Members of this class are mutable.

⇒ `:object` [*Initarg*]
⇒ `:type` [*Initarg*]
⇒ `:view` [*Initarg*]
⇒ `:single-box` [*Initarg*]
⇒ `:modifier` [*Initarg*]

All presentation classes must handle these five initargs, which are used to specify, respectively, the object, type, view, single-box, and modifier components of a presentation.

23.2.1 The Presentation Protocol

The following functions comprise the presentation protocol. All classes that inherit from `presentation` must implement methods for these generic functions.

⇒ `presentation-object` *presentation* [*Generic Function*]

Returns the object associated with the *presentation presentation*.

⇒ `(setf presentation-object)` *object presentation* [*Generic Function*]

Changes the object associated with the *presentation presentation* to *object*.

⇒ `presentation-type` *presentation* [*Generic Function*]

Returns the presentation type associated with the *presentation presentation*.

⇒ `(setf presentation-type)` *type presentation* [*Generic Function*]

Changes the object associated with the *presentation presentation* to *object*.

⇒ `presentation-single-box` *presentation* [*Generic Function*]

Returns the "single box" attribute of the *presentation presentation*, which controls how the presentation is highlighted and when it is sensitive. This will be one of four values:

- `nil` (the default)—if the pointer is pointing at a visible piece of the output that was drawn as part of the presentation, then it is considered to be pointing at the presentation. The presentation is highlighted by highlighting each visible part of the output that was drawn as part of the presentation.
- `t`—if the pointer is inside the bounding rectangle of the presentation, it is considered to be pointing at the presentation. The presentation is highlighted by drawing a thin border around the bounding rectangle.
- `:position`—like `t` for determining whether the pointer is pointing at the presentation, but like `nil` for highlighting.
- `:highlighting`—like `nil` for determining whether the pointer is pointing at the presentation, but like `t` for highlighting.

⇒ `(setf presentation-single-box)` *single-box presentation* [*Generic Function*]

Changes the "single box" attribute of the *presentation presentation* to *single-box*.

⇒ `presentation-modifier` *presentation* [*Generic Function*]

Returns the "modifier" associated with the *presentation presentation*. The modifier is some sort of object that describes how the presentation object might be modified. For example, it might be a function of one argument (the new value) that can be called in order to store a new value for *object* after a user somehow "edits" the presentation.

23.3 Presentation Types

The type associated with a presentation is specified with a *presentation type specifier*, an object matching one of the following three patterns:

```
name
(name parameters...)
((name parameters...)  options...)
```

Note that *name* can be either a symbol that names a presentation type or a CLOS class object (but not a `built-in-class` object), in order to support anonymous CLOS classes.

The *parameters* "parameterize" the type, just as in a Common Lisp type specifier. The function `presentation-typep` uses the parameters to check object membership in a type. Adding parameters to a presentation type specifier produces a subtype, which contains some, but not necessarily all, of the objects that are members of the unparameterized type. Thus the parameters can turn off the sensitivity of some presentations that would otherwise be sensitive.

The *options* are alternating keywords and values that affect the use or appearance of the presentation, but not its semantic meaning. The *options* have no effect on presentation sensitivity. (A programmer could choose to make a tester in a translator examine options, but this is not standard practice.) The standard option `:description` is accepted by all types; if it is a non-`nil` value, then the value must be a string that describes the type and overrides the description supplied by the type's definition.

Every presentation type is associated with a CLOS class. If *name* is a class object or the name of a class, and that class is not a `built-in-class`, that class is the associated class. Otherwise, `define-presentation-type` defines a class with metaclass `presentation-type-class` and superclasses determined by the presentation type definition. This class is not named *name*, since that could interfere with built-in Common Lisp types such as `and`, `member`, and `integer`. `class-name` of this class returns a list (`presentation-type` *name*). `presentation-type-class` is a subclass of `standard-class`.

Implementations are permitted to require programmers to evaluate the `defclass` form first in the case when the same name is used in both a `defclass` and a `define-presentation-type`.

Every CLOS class (except for built-in classes) is a presentation type, as is its name. If it has not been defined with `define-presentation-type`, it allows no parameters and no options.

Presentation type inheritance is used both to inherit methods ("what parser should be used for this type?"), and to establish the semantics for the type ("what objects are sensitive in this input context?"). Inheritance of methods is the same as in CLOS and thus depends only on the type name, not on the parameters and options.

During presentation method combination, presentation type inheritance arranges to translate the parameters of a subtype into a new set of parameters for its supertype, and translates the options of the subtype into a new set of options for the supertype.

23.3.1 Defining Presentation Types

⇒ `define-presentation-type` *name parameters* `&key` *options inherit-from description history parameters-are-types* [*Macro*]

Defines a presentation type whose name is the symbol or class *name* and whose parameters are specified by the lambda-list *parameters*. These parameters are visible within *inherit-from* and within the methods created with `define-presentation-method`. For example, the parameters are used by `presentation-typep` and `presentation-subtypep` methods to refine their tests for type inclusion.

options is a list of option specifiers. It defaults to `nil`. An option specifier is either a symbol or a list (*symbol* `&optional` *default supplied-p presentation-type accept-options*), where *symbol*, *default*, and *supplied-p* are as in a normal lambda-list. If *presentation-type* and *accept-options* are present, they specify how to accept a new value for this option from the user. *symbol* can also be specified in the (*keyword variable*) form allowed for Common Lisp lambda lists. *symbol* is a variable that is visible within *inherit-from* and within most of the methods created with `define-presentation-method`. The keyword corresponding to *symbol* can be used as an option in the third form of a presentation type specifier. An option specifier for the standard option `:description` is automatically added to *options* if an option with that keyword is not present, however it does not produce a visible variable binding.

Unsupplied optional or keyword parameters default to `*` (as in `deftype`) if no default is specified in *parameters*. Unsupplied options default to `nil` if no default is specified in *options*.

inherit-from is a form that evaluates to a presentation type specifier for another type from which the new type inherits. *inherit-from* can access the parameter variables bound by the *parameters* lambda list and the option variables specified by *options*. If *name* is or names a CLOS class (other than a `built-in-class`), then *inherit-from* must specify the class's direct superclasses (using `and` to specify multiple inheritance). It is useful to do this when you want to parameterize previously defined CLOS classes.

If *inherit-from* is unsupplied, it defaults as follows: If *name* is or names a CLOS class, then the type inherits from the presentation type corresponding to the direct superclasses of that CLOS class (using `and` to specify multiple inheritance). Otherwise, the type named by *name* inherits from `standard-object`.

description is a string or `nil`. This should be the term for an instance for the type being defined. If it is `nil` or unsupplied, a description is automatically generated; it will be a "prettied up" version of the type name, for example, `small-integer` would become `"small integer"`. You can also write a `describe-presentation-type` presentation method. *description* is implemented by the default `describe-presentation-type` method, so *description* only works in presentation types where that default method is not shadowed.

history can be `t` (the default), which means this type has its own history of previous inputs, `nil`, which means this type keeps no history, or the name of another presentation type, whose history is shared by this type. More complex histories can be specified by writing a `presentation-type-history` presentation method.

Minor issue: *What is a presentation type history? Should they be exposed? — SWM*

If the boolean *parameters-are-types* is *true*, this means that the parameters to the presentation type are themselves presentation types. If they are not presentation types, *parameters-are-types* should be supplied as *false*. Types such as `and`, `or`, and `sequence` will specify this as *true*.

Every presentation type must define or inherit presentation methods for `accept` and `present` if the type is going to be used for input and output. For presentation types that are only going to be used for input via the pointer, the `accept` need not be defined.

If a presentation type has *parameters*, it must define presentation methods for `presentation-typep` and `presentation-subtypep` that handle the parameters, or inherit appropriate presentation methods. In many cases it should also define presentation methods for `describe-presentation-type` and `presentation-type-specifier-p`.

There are certain restrictions on the *inherit-from* form, to allow it to be analyzed at compile time. The form must be a simple substitution of parameters and options into positions in a fixed framework. It cannot involve conditionals or computations that depend on valid values for the parameters or options; for example, it cannot require parameter values to be numbers. It cannot depend on the dynamic or lexical environment. The form will be evaluated at compile time with uninterned symbols used as dummy values for the parameters and options. In the type specifier produced by evaluating the form, the type name must be a constant that names a type, the type parameters cannot derive from options of the type being defined, and the type options cannot derive from parameters of the type being defined. All presentation types mentioned must be already defined. `and` can be used for multiple inheritance, but `or`, `not`, and `satisfies` cannot be used.

None of the arguments, except *inherit-from*, is evaluated.

23.3.2 Presentation Type Abbreviations

⇒ `define-presentation-type-abbreviation` *name parameters equivalent-type* `&key` *options* [*Macro*]

name, *parameters*, and *options* are as in `define-presentation-type`. This defines a presentation type that is an *abbreviation* for the presentation type *equivalent-type*. Presentation type abbreviations can only be used in places where this specification explicitly permits them. In such places, *equivalent-type* and *abbreviation* are exactly equivalent and can be used interchangeably.

name must be a symbol and must not be the name of a CLOS class.

The *equivalent-type* form might be evaluated at compile time if presentation type abbreviations are expanded by compiler optimizers. Unlike *inherit-from*, *equivalent-type* can perform arbitrary computations and is not called with dummy parameter and option values. The type specifier produced by evaluating *equivalent-type* can be a real presentation type or another abbreviation. If the type specifier doesn't include the standard option `:description`, the option is automatically copied from the abbreviation to its expansion.

Note that you cannot define any presentation methods on a presentation type abbreviation. If you need methods, use `define-presentation-type` instead.

`define-presentation-type-abbreviation` is used to name a commonly used cliche. For example, a presentation type to read an octal integer might be defined as

```
(define-presentation-type-abbreviation octal-integer (&optional low high)
    `((integer ,low ,high) :base 8 :description "octal integer"))
```

None of the arguments, except *equivalent-type*, is evaluated.

⇒ `expand-presentation-type-abbreviation-1` *type* `&optional` *env* [*Function*]

If the *presentation type specifier type* is a presentation type abbreviation, or is an `and`, `or`, `sequence`, or `sequence-enumerated` that contains a presentation type abbreviation, then this expands the type abbreviation once, and returns two values, the expansion and `t`. If *type* is not a presentation type abbreviation, then the values *type* and `nil` are returned.

env is a macro-expansion environment, as for `macroexpand`.

⇒ `expand-presentation-type-abbreviation` *type* `&optional` *env* [*Function*]

`expand-presentation-type-abbreviation` is like `expand-presentation-type-abbreviation-1`, except that *type* is repeatedly expanded until all presentation type abbreviations have been removed.

23.3.3 Presentation Methods

Presentation methods inherit and combine in the same way as ordinary CLOS methods. The reason presentation methods are not exactly the same as ordinary CLOS methods revolves around the *type* argument. The parameter specializer for *type* is handled in a special way, and presentation method inheritance "massages" the type parameters and options seen by each method. For example, consider three types `int`, `rrat`, and `num` defined as follows:

Minor issue: *How are massaged arguments passed along? Right now, we pass along those parameters of the same name, and no others. — SWM*

```
(define-presentation-type int (low high)
  :inherit-from `(rrat ,high ,low))

(define-presentation-method presentation-typep :around (object (type int))
  (and (call-next-method)
       (integerp object)
       (<= low object high)))

(define-presentation-type rrat (high low)
  :inherit-from 'num)

(define-presentation-method presentation-typep :around (object (type rrat))
  (and (call-next-method)
       (rationalp object)
       (<= low object high)))
```

```
(define-presentation-type num ())

(define-presentation-method presentation-typep (object (type num))
  (numberp object))
```

If the user were to evaluate the form `(presentation-typep X '(int 1 5))`, then the type parameters will be `(1 5)` in the `presentation-typep` method for `int`, `(5 1)` in the method for `rrat`, and `nil` in the method for `num`. The value for *type* will be or `((int 1 5))` in each of the methods.

⇒ `define-presentation-generic-function` *generic-function-name presentation-function-name lambda-list* `&rest` *options* [*Macro*]

Defines a generic function that will be used for presentation methods. *generic-function-name* is a symbol that names the generic function that will be used internally by CLIM for the individual methods, *presentation-function-name* is a symbol that names the function that programmers will call to invoke the method, and *lambda-list* and *options* are as for `defgeneric`.

There are some "special" arguments in *lambda-list* that are known about by the presentation type system. The first argument in *lambda-list* must be either `type-key` or `type-class`; this argument is used by CLIM to implement method dispatching. The second argument may be `parameters`, meaning that, when the method is invoked, the type parameters will be passed to it. The third argument may be `options`, meaning that, when the method is invoked, the type options will be passed to it. Finally, an argument named `type` must be included in *lambda-list*; when the method is called, *type* argument will be bound to the presentation type specifier.

For example, the `accept` presentation generic function might be defined as follows:

```
(define-presentation-generic-function present-method present
  (type-key parameters options object type stream view
   &key acceptably for-context-type))
```

None of the arguments is evaluated.

⇒ `define-presentation-method` *name qualifiers* specialized-lambda-list* `&body` *body* [*Macro*]

Defines a presentation method for the function named *name* on the presentation type named in *specialized-lambda-list*. *specialized-lambda-list* is a CLOS specialized lambda list for the method, and its contents varies depending on what *name* is. *qualifiers** is zero or more of the usual CLOS method qualifier symbols. `define-presentation-method` must support at least `standard` method combination (and therefore the `:before`, `:after`, and `:around` method qualifiers). Some CLIM implementations may support other method combination types, but this is not required.

body defines the body of the method. *body* may have zero or more declarations as its first forms.

All presentation methods have an argument named *type* that must be specialized with the name of a presentation type. The value of *type* is a presentation type specifier, which can be for a subtype that inherited the method.

All presentation methods except `presentation-subtypep` have lexical access to the parameters from the presentation type specifier. Presentation methods for the functions `accept`, `present`, `describe-presentation-type`, `presentation-type-specifier-p`, and `accept-present-default` also have lexical access to the options from the presentation type specifier.

⇒ `define-default-presentation-method` *name qualifiers* specialized-lambda-list* `&body` *body* [*Macro*]

Like `define-presentation-method`, except that it is used to define a default method that will be used only if there are no more specific methods.

⇒ `funcall-presentation-generic-function` *presentation-function-name* `&rest` *arguments* [*Macro*]

Calls the presentation generic function named by *presentation-function-name* on the arguments *arguments*. *arguments* must match the arguments specified by the `define-presentation-generic-function` that was used to define the presentation generic function, excluding the `type-key`, `type-class`, `parameters`, and `options` arguments, which are filled in by CLIM.

`funcall-presentation-generic-function` is analogous to `funcall`.

The *presentation-function-name* argument is not evaluated.

For example, to call the `present` presentation generic function, one might use the following:

```
(funcall-presentation-generic-function present
  object presentation-type stream view)
```

⇒ `apply-presentation-generic-function` *presentation-function-name* `&rest` *arguments* [*Macro*]

Like `funcall-presentation-generic-function`, except that `apply-presentation-generic-function` is analogous to `apply`.

The *presentation-function-name* argument is not evaluated.

Here is a list of all of the standard presentation methods and their specialized lambda lists. For the meaning of the arguments to each presentation method, refer to the description of the function that calls that method.

For all of the presentation methods, the *type* will always be specialized. For those methods that take a *view* argument, implementors and programmers may specialize it as well. The other arguments are not typically specialized.

⇒ `present` *object type stream view* `&key` *acceptably for-context-type* [*Presentation Method*]

The `present` presentation method is responsible for displaying the representation of *object* having *presentation type type* for a particular *view view*. The method's caller takes care of creating the presentation, the method simply displays the content of the presentation.

The `present` method can specialize on the *view* argument in order to define more than one view of the data. For example, a spreadsheet program might define a presentation type for revenue, which can be displayed either as a number or a bar of a certain length in a bar graph. Typically, at least one canonical view should be defined for a presentation type, for example, the `present` method for the `textual-view` view must be defined if the programmer wants to allow objects of that type to be displayed textually.

Implementation note: the actual argument list to the `present` method is
(type-key parameters options object type stream view `&key` *acceptably for-context-type)*
type-key is the object that is used to cause the appropriate methods to be selected (an instance of the class that corresponds to the presentation type *type*.). *parameters* and *options* are the parameters and options for the type on which the current method is specialized. The other arguments are gotten from the arguments of the same name in `present`.

Implementation note: the actual generic function of the `present` method is an internal generic function, not the function whose name is `present`. Similar internal generic functions are used for all presentation methods.

⇒ `accept` *type stream view* `&key` *default default-type* [*Presentation Method*]

The `accept` method is responsible for "parsing" the representation of the *presentation type type* for a particular *view view*. The `accept` method must return a single value, the object that was "parsed", or two values, the object and its type (a presentation type specifier). The method's caller takes care of establishing the input context, defaulting, prompting, and input editing.

The `accept` method can specialize on the *view* argument in order to define more than one input view for the data. The `accept` method for the `textual-view` view must be defined if the programmer wants to allow objects of that type to entered via the keyboard.

Note that `accept` presentation methods can call `accept` recursively. In this case, the programmer should be careful to specify `nil` for `:prompt` and `:display-default` unless recursive prompting is really desired.

Implementation note: the actual argument list to the `accept` method is
(type-key parameters options type stream view `&key` *default default-type)*

⇒ `describe-presentation-type` *type stream plural-count* [*Presentation Method*]

The `describe-presentation-type` method is responsible for textually describing the *presentation type type*. *stream* is a stream, and will not be `nil` as it can be for the `describe-presentation-type` function.

Implementation note: the actual argument list to the `describe-presentation-type` method is
(type-key parameters options type stream plural-count)

⇒ `presentation-type-specifier-p` *type* [*Presentation Method*]

The `presentation-type-specifier-p` method is responsible for checking the validity of the parameters and options for the *presentation type type*. The default method returns `t`.

Implementation note: the actual argument list to the `presentation-type-specifier-p` method is
(type-key parameters options type)

⇒ `presentation-typep` *object type* [*Presentation Method*]

The `presentation-typep` method is called when the `presentation-typep` function requires type-specific knowledge. If the type name in the *presentation type type* is a CLOS class or names a CLOS class, the method is called only if *object* is a member of the class and *type* contains parameters, and the method simply tests whether *object* is a member of the subtype specified by the parameters. For non-class types, the method is always called.

Implementation note: the actual argument list to the `presentation-typep` method is
(type-key parameters object type)

⇒ `presentation-subtypep` *type putative-supertype* [*Presentation Method*]

`presentation-subtypep` walks the type lattice (using `map-over-presentation-supertypes`) to determine whethe or not the *presentation type type* is a subtype of the*presentation type putative-supertype*, without looking at the type parameters. When a supertype of *type* has been found whose name is the same as the name of *putative-supertype*, then the `subtypep` method for that type is called in order to resolve the question by looking at the type parameters (that is, if the `subtypep` method is called, *type* and *putative-supertype* are guaranteed to be the same type, differing only in their parameters). If *putative-supertype* is never found during the type walk, then `presentation-subtypep` will never call the `presentation-subtypep` presentation method for *putative-supertype*.

Unlike all other presentation methods, `presentation-subtypep` receives a *type* argument that has been translated to the presentation type for which the method is specialized; *type* is never a subtype. The method is only called if *putative-supertype* has parameters and the two presentation type specifiers do not have equal parameters. The method must return the two values that `presentation-subtypep` returns.

Since `presentation-subtypep` takes two type arguments, the parameters are not lexically available as variables in the body of a presentation method.

Implementation note: the actual argument list to the `presentation-subtypep` method is
(type-key type putative-supertype)

⇒ `map-over-presentation-type-supertypes` *function type* [*Presentation Method*]

This method is called in order to apply *function* to the superclasses of the *presentation type type*.

Implementation note: the actual argument list to the `map-over-presentation-type-supertypes` method is
(type-class function type)

⇒ `accept-present-default` *type stream view default default-supplied-p present-p query-identifier* [*Presentation Method*]

The `accept-present-default` method is called when `accept` turns into `present` inside of `accepting-values`. The default method calls `present` or `describe-presentation-type` depending on whether *default-supplied-p* is *true* or *false*, respectively.

type, *stream*, *view*, *default*, and *query-identifier* are as for `accept`. *present-p* is a list whose first element is the presentation type of the "query" corresponding to the dialog field, and whose second element is the query itself. `accepting-values` is discussed in detail in Chapter 26.

The boolean *default-supplied-p* will be *true* only in the case when the `:default` option was explicitly supplied in the call to `accept` that invoked `accept-present-default`.

Implementation note: the actual argument list to the `accept-present-default` method is *(type-key parameters options type stream view default default-supplied-p present-p query-identifier)*

⇒ `presentation-type-history` *type* [*Presentation Method*]

This method is responsible for returning a history object for the *presentation type type*.

Implementation note: the actual argument list to the `presentation-type-history` method is
(type-key parameters type)

⇒ `presentation-default-preprocessor` *default type* `&key` *default-type* [*Presentation Method*]

This method is responsible for taking the object *default*, and coercing it to match the *presentation type type* (which is the type being accepted) and *default-type* (which is the presentation type of *default*). This is useful when you want to change the default gotten from the presentation type's history so that it conforms to parameters or options in *type* and *default-type*.) The method must return two values, the new object to be used as the default, and a new presentation type, which should be at least as specific as *type*.

Implementation note: the actual argument list to the `presentation-default-preprocessor` method is
(type-key parameters default type `&key` *default-type)*

⇒ `presentation-refined-position-test` *type record x y* [*Presentation Method*]

This method used to definitively answer hit detection queries for a presentation, that is, determining that the point (x, y) is contained within the output record *record*. Its contract is exactly the same as for `output-record-refined-position-test`, except that it is intended to specialize on the presentation type *type*.

Implementation note: the actual argument list to the `presentation-refined-position-test` method is
(type-key parameters options type record x y)

⇒ `highlight-presentation` *type record stream state* [*Presentation Method*]

This method is responsible for drawing a highlighting box around the *presentation record* on the *output recording stream stream*. *state* will be either `:highlight` or `:unhighlight`.

Implementation note: the actual argument list to the `highlight-presentation` method is *(type-key parameters options type record stream state)*

23.3.4 Presentation Type Functions

⇒ `describe-presentation-type` *type* `&optional` *stream plural-count* [*Function*]

Describes the *presentation type specifier type* on the *stream stream*, which defaults to `*standard-output*`. If *stream* is `nil`, a string containing the description is returned. *plural-count* is either `nil` (meaning that the description should be the singular form of the name), `t` (meaning that the description should the plural form of the name), or an integer greater than zero (the number of items to be described). The default is `1`.

type can be a presentation type abbreviation.

⇒ `presentation-type-parameters` *type-name* `&optional` *env* [*Function*]

Returns a lambda-list, the parameters specified when the presentation type or presentation type abbreviation whose name is *type-name* was defined. *type-name* is a symbol or a class. *env* is a macro-expansion environment, as in `find-class`.

⇒ `presentation-type-options` *type-name* `&optional` *env* [*Function*]

Returns the list of options specified when the presentation type or presentation type abbreviation whose name is *type-name* was defined. This does not include the standard options unless the presentation-type definition mentioned them explicitly. *type-name* is a symbol or a class. *env* is a macro-expansion environment, as in `find-class`.

⇒ `with-presentation-type-decoded` *(name-var* `&optional` *parameters-var options-var) type* `&body` *body* [*Macro*]

The specified variables are bound to the components of the presentation type specifier produced by evaluating *type*, the forms in *body* are executed, and the values of the last form are returned. *name-var*, if non-`nil`, is bound to the presentation type name. *parameters-var*, if non-`nil`, is bound to a list of the parameters. *options-var*, if non-`nil`, is bound to a list of the options. When supplied, *name-var*, *parameters-var*, and *options-var* must be symbols.

The *name-var*, *parameters-var*, and *options-var* arguments are not evaluated. *body* may have zero or more declarations as its first forms.

⇒ `presentation-type-name` *type* [*Function*]

Returns the presentation type name of the presentation type specifier *type*. This function is provided as a convenience. It could be implemented with the following code:

```
(defun presentation-type-name (type)
  (with-presentation-type-decoded (name) type
    name))
```

⇒ `with-presentation-type-parameters` *(type-name type)* `&body` *body* [*Macro*]

Variables with the same name as each parameter in the definition of the presentation type are bound to the parameter values in *type*, if present, or else to the defaults specified in the definition of the presentation type. The forms in *body* are executed in the scope of these variables and the values of the last form are returned.

The value of the form *type* must be a presentation type specifier whose name is *type-name*. The *type-name* and *type* arguments are not evaluated. *body* may have zero or more declarations as its first forms.

⇒ `with-presentation-type-options` *(type-name type)* `&body` *body* [*Macro*]

Variables with the same name as each option in the definition of the presentation type are bound to the option values in *type*, if present, or else to the defaults specified in the definition of the presentation type. The forms in *body* are executed in the scope of these variables and the values of the last form are returned.

The value of the form *type* must be a presentation type specifier whose name is *type-name*. The *type-name* and *type* arguments are not evaluated. *body* may have zero or more declarations as its first forms.

⇒ `presentation-type-specifier-p` *object* [*Function*]

Returns *true* if *object* is a valid *presentation type specifier*, otherwise returns *false*.

⇒ `presentation-typep` *object type* [*Function*]

Returns *true* if *object* is of the presentation type specified by the *presentation type specifier type*, otherwise returns *false*.

type may not be a presentation type abbreviation.

This is analogous to the Common Lisp `typep` function.

⇒ `presentation-type-of` *object* [*Function*]

Returns a presentation type of which *object* is a member. `presentation-type-of` returns the most specific presentation type that can be conveniently computed and is likely to be useful to the programmer. This is often the class name of the class of the object.

If `presentation-type-of` cannot determine the presentation type of the object, it may return either `expression` or `t`.

This is analogous to the Common Lisp `typep` function.

⇒ `presentation-subtypep` *type putative-supertype* [*Function*]

Answers the question "is the type specified by the *presentation type specifier type* a subtype of the type specified by the *presentation type specifier putative-supertype*?". `presentation-subtypep` returns two values, *subtypep* and *known-p*. When *known-p* is *true*, *subtypep* can be either *true* (meaning that *type* is definitely a subtype of *putative-supertype*) or *false* (meaning that *type* is definitely not a subtype of *putative-supertype*). When *known-p* is *false*, then *subtypep* must also be *false*; this means that the answer cannot reliably be determined.

type may not be a presentation type abbreviation.

This is analogous to the Common Lisp `subtypep` function.

⇒ `map-over-presentation-type-supertypes` *function type* [*Function*]

Calls the function *function* on the *presentation type specifier type* and each of its supertypes. *function* is called with two arguments, the name of a type and a presentation type specifier for that type with the parameters and options filled in. *function* has dynamic extent; its two arguments are permitted to have dynamic extent. The traversal of the type lattice is done in the order specified by the CLOS class precedence rules, and visits each type in the lattice exactly once.

`map-over-presentation-type-supertypes` returns `nil`.

⇒ `presentation-type-direct-supertypes` *type* [*Function*]

Returns a sequence consisting of the names of all of the presentation types that are direct supertypes of the *presentation type specifier type*, or `nil` if *type* has no supertypes. The consequences of modifying the returned sequence are unspecified.

⇒ `find-presentation-type-class` *name* `&optional` *(errorp* `t`*) environment* [*Function*]

Returns the class corresponding to the presentation type named *name*, which must be a symbol or a class object. *errorp* and *environment* are as for `find-class`.

⇒ `class-presentation-type-name` *class* `&optional` *environment* [*Function*]

Returns the presentation type name corresponding to the class *class*. This is essentially the inverse of `find-presentation-type-class`. *environment* is as for `find-class`.

⇒ `default-describe-presentation-type` *description stream plural-count* [*Function*]

Performs the default actions for `describe-presentation-type`, notably pluralization and prepending an indefinite article if appropriate. *description* is a string or a symbol, typically the `:description` presentation type option or the `:description` option to `define-presentation-type`. *plural-count* is as for `describe-presentation-type`.

⇒ `make-presentation-type-specifier` *type-name-and-parameters* `&rest` *options* [*Function*]

A convenient way to assemble a presentation type specifier with only non-default options included. This is only useful for abbreviation expanders, not for `:inherit-from`. *type-name-and-parameters* is a presentation type specifier, which must be in the (*type-name parameters...*) form. *options* are alternating keywords and values that are added as options to the presentation type specifier, except that if a value is equal to *type-name*'s default, that option is omitted, producing a more concise presentation type specifier.

23.4 Typed Output

An application can specify that all output done within a certain dynamic extent should be associated with a given Lisp object and be declared to be of a specified presentation type. The resulting output is saved in the window's output history as a presentation. Specifically, the presentation remembers the output that was performed (by saving the associated output record), the Lisp object associated with the output, and the presentation type specified at output time. The object can be any Lisp object.

⇒ `with-output-as-presentation` *(stream object type* `&key` *modifier single-box allow-sensitive-inferiors parent record-type* `&allow-other-keys` *)* `&body` *body* *[Macro]*

The output of *body* to the *extended output recording stream* is used to generate a presentation whose underlying object is *object* and whose presentation type is *type*. Each invocation of this macro results in the creation of a presentation object in the stream's output history unless output recording has been disabled or `:allow-sensitive-inferiors nil` was specified at a higher level, in which case the presentation object is not inserted into the history. `with-output-as-presentation` returns the presentation corresponding to the output.

The *stream* argument is not evaluated, and must be a symbol that is bound to an extended output stream or output recording stream. If *stream* is `t`, `*standard-output*` is used. *body* may have zero or more declarations as its first forms.

type may be a presentation type abbreviation.

modifier, which defaults to `nil`, is some sort of object that describes how the presentation object might be modified. For example, it might be a function of one argument (the new value) that can be called in order to store a new value for *object* after a user somehow "edits" the presentation. *modifier* must have indefinite extent.

single-box is used to specify the `presentation-single-box` component of the resulting presentation. It can take on the values described under `presentation-single-box`.

When the boolean *allow-sensitive-inferiors* is *false*, nested calls to `present` or `with-output-as-presentation` inside this one will not generate presentations. The default is *true*.

parent specifies what output record should serve as the parent for the newly created presentation. If unspecified, `stream-current-output-record` of *stream* will be used as the parent.

record-type specifies the class of the presentation output record to be created. It defaults to `standard-presentation`. This argument should only be supplied by a programmer if there is a new class of output record that supports the updating output record protocol.

All arguments of this macro are evaluated.

For example,

```
(with-output-as-presentation (stream #p"foo" 'pathname)
  (princ "FOO" stream))
```

⇒ `present` *object* `&optional` *type* `&key` *stream view modifier acceptably for-context-type single-box allow-sensitive-inferiors sensitive record-type* [*Function*]

The *object* of *presentation type type* is presented to the *extended output stream stream* (which defaults to `*standard-output*`), using the type's `present` method for the supplied *view view*. *type* is a presentation type specifier, and can be an abbreviation. It defaults to `(presentation-type-of` *object*`)`. The other arguments and overall behavior of `present` are as for `stream-present`.

The returned value of `present` is the presentation object that contains the output corresponding to the object.

`present` must be implemented by first expanding any presentation type abbreviations (*type* and *for-context-type*), and then calling `stream-present` on *stream*, *object*, *type*, and the remaining keyword arguments, which are described below.

⇒ `stream-present` *stream object type* `&key` *view modifier acceptably for-context-type single-box allow-sensitive-inferiors sensitive record-type* [*Generic Function*]

`stream-present` is the per-stream implementation of `present`, analogous to the relationship between `write-char` and `stream-write-char`. All extended output streams and output recording streams must implement a method for `stream-present`. The default method (on `standard-extended-output-stream`) implements the following behavior.

The object *object* of type *type* is presented to the *stream stream* by calling the type's `present` method for the supplied *view view*. The returned value is the presentation containing the output corresponding to the object.

type is a presentation type specifier. *view* is a view object that defaults to `stream-default-view` of *stream*.

for-context-type is a presentation type specifier that is passed to the `present` method for *type*, which can use it to tailor how the object will be presented. *for-context-type* defaults to *type*.

modifier, *single-box*, *allow-sensitive-inferiors*, and *record-type* are the same as for `with-output-as-presentation`.

acceptably defaults to `nil`, which requests the `present` method to produce text designed to be read by human beings. If *acceptably* is `t`, it requests the `present` method to produce text that is recognized by the `accept` method for *for-context-type*. This makes no difference to most presentation types.

The boolean *sensitive* defaults to *true*. If it is *false*, no presentation is produced.

⇒ `present-to-string` *object* `&optional` *type* `&key` *view acceptably for-context-type string index*

[*Function*]

Same as `present` inside `with-output-to-string`. If *string* is supplied, it must be a string with a fill pointer. When *index* is supplied, it is used as an index into *string*. *view*, *acceptably*, and *for-context-type* are as for `present`.

The first returned value is the string. When *string* is supplied, a second value is returned, the updated *index*.

23.5 Context-dependent (Typed) Input

Associating semantics with output is only half of the user interface equation. The presentation type system also supports the input side of the user interaction. When an application wishes to solicit from the user input of a particular presentation type, it establishes an *input context* for that type. CLIM will then automatically allow the user to satisfy the input request by pointing at a visible presentation of the requested type (or a valid subtype) and pressing a pointer button. Only the presentations that "match" the input context will be "sensitive" (that is, highlighted when the pointer is moved over them) and accepted as input, thus the presentation-based input mechanism supports *context-dependent input*.

Minor issue: *What exactly is an input context? What does it mean for them to be nested? — SWM*

⇒ `*input-context*` [*Variable*]

The current input context. This will be a list, each element of which corresponds to a single call to `with-input-context`. The first element of the list represents the context established by the most recent call to `with-input-context`, and the last element represents the context established by the least recent call to `with-input-context`.

The exact format of the elements in the list is unspecified, but will typically be a list of a presentation type and a tag that corresponds to the point in the control structure of CLIM at which the input context was establish. `*input-context*` and the elements in it may have dynamic extent.

⇒ `input-context-type` *context-entry* [*Function*]

Given one element from `*input-context*`, *context-entry*, returns the presentation type of the context entry.

⇒ `with-input-context` *(type* `&key` *override)* (`&optional` *object-var type-var event-var options-var) form* `&body` *pointer-cases* [*Macro*]

Establishes an input context of *presentation type type*; this must be done by binding `*input-context*` to reflect the new input context. When the boolean *override* is *false* (the default), this invocation of `with-input-context` adds its context presentation type to the current context. In this way an application can solicit more than one type of input at the same time. When *override* is *true*, it overrides the current input context rather than nesting inside the current input context.

type can be a presentation type abbreviation.

After establishing the new input context, *form* is evaluated. If no pointer gestures are made by the user during the evaluation of *form*, the values of *form* are returned. Otherwise, one of the *pointer-cases* is executed (based on the presentation type of the object that was clicked on) and the value of that is returned. (See the descriptions of `call-presentation-menu` and `throw-highlighted-presentation`.) *pointer-cases* is constructed like a `typecase` statement clause list whose keys are presentation types; the first clause whose key satisfies the condition (`presentation-subtypep` *type key*) is the one that is chosen.

During the execution of one of the *pointer-cases*, *object-var* is bound to the object that was clicked on (the first returned value from the presentation translator that was invoked), *type-var* is bound to its presentation type (the second returned value from the translator), and *event-var* is bound to the pointer button event that was used. *options-var* is bound to any options that a presentation translator might have returned (the third value from the translator), and will be either `nil` or a list of keyword-value pairs. *object-var*, *type-var*, *event-var*, and *options-var* must all be symbols.

type, *stream*, and *override* are evaluated, the others are not.

For example,

```
(with-input-context ('pathname)
                    (path)
     (read)
   (pathname
     (format t "~&The pathname ~A was clicked on." path)))
```

⇒ `accept` *type* `&key` *stream view default default-type provide-default insert-default replace-input history active-p prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures* [*Function*]

Requests input of type *type* from the *stream stream*, which defaults to `*standard-input*`. `accept` returns two values, the object representing the input and its presentation type. *type* is a presentation type specifier, and can be an abbreviation. The other arguments and overall behavior of `accept` are as for `accept-1`.

`accept` must be implemented by first expanding any presentation type abbreviations (*type*, *default-type*, and *history*), handling the interactions between the default, default type, and presentation history, prompting the user by calling `prompt-for-accept`, and then calling `stream-accept` on *stream*, *type*, and the remaining keyword arguments.

⇒ `stream-accept` *stream type* `&key` *view default default-type provide-default insert-default replace-input history active-p prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures* [*Generic Function*] `stream-accept` is the per-stream implementation of `accept`, analogous to the relationship between `read-char` and `stream-read-char`. All extended input streams must implement a method for `stream-accept`. The default method (on `standard-extended-input-stream`) simply calls `accept-1`.

The arguments and overall behavior of `stream-accept` are as for `accept-1`.

Rationale: the reason `accept` is specified as a three-function "trampoline" is to allow close tailoring of the behavior of `accept`. `accept` itself is the function that should be called by application programmers. CLIM implementors will specialize `stream-accept` on a per-stream basis. (For example, the behavior of `accepting-values` can be implemented by creating a special class of stream that turns calls to `accept` into fields of a dialog.) `accept-1` is provided as a convenient function for the `stream-accept` methods to call when they require the default behavior.

⇒ `accept-1` *stream type* `&key` *view default default-type provide-default insert-default replace-input history active-p prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures* [*Function*]

Requests input of type *type* from the *stream stream*. *type* must be a presentation type specifier. *view* is a view object that defaults to `stream-default-view` of *stream*. `accept-1` returns two values, the object representing the input and its presentation type. (If `frame-maintain-presentation-histories` is *true* for the current frame, then the returned object is also pushed on to the presentation history for that object.)

`accept-1` establishes an input context via `with-input-context`, and then calls the `accept` presentation method for *type* and *view* (except when inside of calls to `accepting-values`). When called on an interactive stream, `accept` must allow input editing; see Chapter 24 for a discussion of input editing. The call to `accept` will be terminated when the `accept` method returns, or the user clicks on a sensitive presentation. The typing of an activation and delimiter character is typically one way in which a call to an `accept` method is terminated.

When `accept-1` is called inside of a call to `accepting-values`, it will call the `accept-present-default` presentation method instead of the `accept` presentation method. In this case, `accept-1` will return the values returned by `accept-present-default`.

A top-level `accept` satisfied by keyboard input discards the terminating keyboard gesture (which will be either a delimiter or an activation gesture). A nested call to `accept` leaves the terminating gesture unread.

If the user clicked on a matching presentation, `accept-1` will insert the object into the input buffer by calling `presentation-replace-input` on the object and type returned by the presentation translator, unless either the boolean *replace-input* is *false* or the presentation translator returned an `:echo` option of *false*. *replace-input* defaults to *true*, but this default is overridden by the translator explicitly returning an `:echo` option of *false*.

If *default* is supplied, then it and *default-type* are returned as values from `accept-1` when the input is empty. *default-type* must be a presentation type specifier. If *default* is not supplied and *provide-default* is *true* (the default is *false*), then the default is determined by taking the most recent item from the presentation type history specified by *history*. If *insert-default* is *true* and there is a default, the default will be inserted into the input stream by calling `presentation-replace-input`.

history must be either `nil`, meaning that no presentation type history will be used, or a presentation type (or abbreviation) that names a history to be used for the call to `accept`. *history* defaults to *type*.

prompt can be `t`, which prompts by describing the type, `nil`, which suppresses prompting, or a string, which is displayed as a prompt (via `write-string`). The default is `t`, which produces "Enter a *type*:" in a top-level call to `accept` or "(*type*)" in a nested call to `accept`.

If the boolean *display-default* is *true*, the default is displayed (if one was supplied). If *display-default* is *false*, the default is not displayed. *display-default* defaults to *true* if *prompt* was provided, otherwise it defaults to *false*.

prompt-mode can be `:normal` (the default) or `:raw`, which suppresses putting a colon after the prompt and/or default in a top-level `accept` and suppresses putting parentheses around the prompt and/or default in a nested `accept`.

query-identifier is used within `accepting-values` to identify the field within the dialog. The `active-p` argument (which defaults to `t`) can be used to control whether a field within an `accepting-values` is active; when *false*, the field will not be active, that is, it will not be available for input. Some CLIM implementations will provide a visual cue that the field is inactive, for instance, by "graying out" the field.

activation-gestures is a list of gesture names that will override the current activation gestures (which are stored in `*activation-gestures*`). Alternatively, *additional-activation-gestures* can be supplied to add activation gestures without overriding the current ones. See Chapter 24 for a discussion of activation gestures.

delimiter-gestures is a list of gesture names that will override the current delimiter gestures (which are stored in `*delimiter-gestures*`). Alternatively, *additional-delimiter-gestures* can be supplied to add delimiter gestures without overriding the current ones. See Chapter 24 for a discussion of delimiter gestures.

⇒ `accept-from-string` *type string* `&key` *view default default-type start end* [*Function*]

Like `accept`, except that the input is taken from *string*, starting at the position specified by *start* and ending at *end*. *view*, *default*, and *default-type* are as for `accept`.

`accept-from-string` returns an object and a presentation type (as in `accept`), but also returns a third value, the index at which input terminated.

⇒ `prompt-for-accept` *stream type view* `&rest` *accept-args* `&key` [*Generic Function*]

Called by `accept` to prompt the user for input of *presentation type type* on the *stream stream* for the *view view*. *accept-args* are all of the keyword arguments supplied to `accept`. The default method (on `standard-extended-input-stream`) simply calls `prompt-for-accept-1`.

⇒ `prompt-for-accept-1` *stream type* `&key` *default default-type display-default prompt prompt-mode* `&allow-other-keys` [*Function*]

Prompts the user for input of *presentation type type* on the *stream stream*.

If the boolean *display-default* is *true*, then the default is displayed; otherwise, the default is not displayed. When the default is being displayed, *default* and *default-type* are the taken as the object and presentation type of the default to display. *display-default* defaults to *true* if *prompt* is non-`nil`, otherwise it defaults to *false*.

If *prompt* is `nil`, no prompt is displayed. If it is a string, that string is displayed as the prompt. If *prompt* is `t` (the default), the prompt is generated by calling `describe-presentation-type` to produce a prompt of the form "Enter a *type*:" in a top-level call to `accept`, or "(*type*)" in a nested call to `accept`.

prompt-mode can be `:normal` (the default) or `:raw`, which suppresses putting a colon after the prompt and/or default in a top-level `accept` and suppresses putting parentheses around the prompt and/or default in a nested `accept`.

23.6 Views

`accept` and `present` methods can specialize on the *view* argument in order to define more than one view of the data. For example, a spreadsheet program might define a presentation type for quarterly earnings, which can be displayed as a floating point number or as a bar of some length in a bar graph. These two views might be implemented by specializing the view arguments for the `textual-view` class and the user-defined `bar-graph-view` class.

⇒ `view` [*Protocol Class*]

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

All of the view classes are immutable.

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

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

⇒ `textual-view` [*Class*]

The instantiable class representing all textual views, a subclass of `view`. Presentation methods that apply to a textual view must only do textual input and output (such as `read-char` and `write-string`).

⇒ `textual-menu-view` [*Class*]

The instantiable class that represents the default view that is used inside `menu-choose` for frame managers that are not using a gadget-oriented look and feel. It is a subclass of `textual-view`.

⇒ `textual-dialog-view` [*Class*]

The instantiable class that represents the default view that is used inside `accepting-values` dialogs for frame managers that are not using a gadget-oriented look and feel. It is a subclass of `textual-view`.

⇒ `gadget-view` [*Class*]

The instantiable class representing all gadget views, a subclass of `view`.

⇒ `gadget-menu-view` [*Class*]

The instantiable class that represents the default view that is used inside `menu-choose` for frame managers that are using a gadget-oriented look and feel. It is a subclass of `gadget-view`.

⇒ `gadget-dialog-view` [*Class*]

The instantiable class that represents the default view that is used inside `accepting-values` dialogs for frame managers that are using a gadget-oriented look and feel. It is a subclass of `gadget-view`.

⇒ `pointer-documentation-view` [*Class*]

The instantiable class that represents the default view that is used when computing pointer documentation. It is a subclass of `textual-view`.

⇒ `+textual-view+` [*Constant*]
⇒ `+textual-menu-view+` [*Constant*]
⇒ `+textual-dialog-view+` [*Constant*]
⇒ `+gadget-view+` [*Constant*]
⇒ `+gadget-menu-view+` [*Constant*]
⇒ `+gadget-dialog-view+` [*Constant*]
⇒ `+pointer-documentation-view+` [*Constant*]

These are objects of class `textual-view`, `textual-menu-view`, `textual-dialog-view`, `gadget-view`, `gadget-menu-view`, `gadget-dialog-view`, and `pointer-documentation-view`, respectively.

⇒ `stream-default-view` *stream* [*Generic Function*]

Returns the default view for the extended stream *stream*. `accept` and `present` get the default value for the *view* argument from this. All extended input and output streams must implement a method for this generic function.

⇒ `(setf stream-default-view)` *view stream* [*Generic Function*]

Changes the default view for *stream* to the *view view*. All extended input and output streams must implement a method for this generic function.

23.7 Presentation Translators

CLIM provides a mechanism for *translating* between types. In other words, within an input context for presentation type A the translator mechanism allows a programmer to define a translation from presentations of some other type B to objects that are of type A.

Note that the exact representation of a presentation translator has been left explicitly unspecified.

23.7.1 Defining Presentation Translators

⇒ `define-presentation-translator` *name (from-type to-type command-table* `&key` *gesture tester tester-definitive documentation pointer-documentation menu priority) arglist* `&body` *body* [*Macro*]

Defines a presentation translator named *name* that translates from objects of type *from-type* to objects of type *to-type*. *from-type* and *to-type* are presentation type specifiers, but must not include any presentation type options. *from-type* and *to-type* may be presentation type abbreviations.

command-table is a *command table designator*. The translator created by this invocation of `define-presentation-translator` will be stored in the command table *command-table*.

gesture is a gesture name that names a pointer gesture (described in Section 22.3). The body of the translator will be run only if the translator is applicable and gesture used by the user matches the gesture name in the translator. (We will explain *applicability*, or *matching*, in detail below.) *gesture* defaults to `:select`. Supplying `:gesture nil` results in a translator that is only available via the `:menu`-gesture menu.

tester is either a function or a list of the form
(tester-arglist . tester-body)
where *tester-arglist* takes the same form as *arglist* (see below), and *tester-body* is the body of the tester. The tester must return either *true* or *false*. If it returns *false*, then the translator is definitely not applicable. If it returns *true*, then the translator might be applicable, and the body of the translator might be run (if *tester-definitive* is *false*) in order to definitively decide if the translator is applicable (this is described in more detail below). If no tester is supplied, CLIM supplies a tester that always returns *true*.

When the boolean *tester-definitive* is *true*, the body of the translator will never be run in order to decide if the translator is applicable, that is, the tester is assumed to definitively decide whether the translator applies. The default for *tester-definitive* is *false*. When there is no explicitly supplied tester, the tester supplied by CLIM is assumed to be definitive.

Both *documentation* and *pointer-documentation* are objects that will be used for documenting the translator. *pointer-documentation* will be used to generate documentation for the pointer documentation window; the documentation generated by *pointer-documentation* should be very brief and computing it should be very fast and preferably not cons. *documentation* is used to generate such things as items in the `:menu`-gesture menu. If the object is a string, the string

itself will be used as the documentation. Otherwise, the object must be the name of a function or a list of the form
(doc-arglist . doc-body)
where *doc-arglist* takes the same form as *arglist*, but includes a named (keyword) *stream* argument as well (see below), and *doc-body* is the body of the documentation function. The body of the documentation function should write the documentation to *stream*. The default for *documentation* is `nil`, meaning that there is no explicitly supplied documentation; in this case, CLIM is free to generate the documentation in other ways. The default for *pointer-documentation* is *documentation*.

menu must be `t` or `nil`. When it is `t`, the translator will be included in the `:menu`-gesture menu if it matches. When it is `nil`, the translator will not be included in the `:menu`-gesture menu. Other non-`nil` values are reserved for future extensions to allow multiple presentation translator menus.

priority is either `nil` (the default, which corresponds to 0) or an integer that represents the priority of the translator. When there are several translators that match for the same gesture, the one with the highest priority is chosen.

arglist, *tester-arglist*, and *doc-arglist* are each an argument list that must "match" the following "canonical" argument list.
(object `&key` *presentation context-type frame event window x y)*
In order to "match" the canonical argument list, there must be a single positional argument that corresponds to the presentation's object, and several named arguments that must match the canonical names above (using `string-equal` to do the comparison).

In the body of the translator (or the tester), the positional *object* argument will be bound to the presentation's object. The named arguments *presentation* will be bound to the presentation that was clicked on, *context-type* will be bound to the presentation type of the context that actually matched, *frame* will be bound to the application frame that is currently active (usually `*application-frame*`), *event* will be bound to the pointer button event that the user used, *window* will be bound to the window stream from which the event came, and *x* and *y* will be bound to the *x* and *y* positions within *window* that the pointer was at when the event occurred. The special variable `*input-context*` will be bound to the current input context. Note that, in many implementations *context-type* and `*input-context*` will have dynamic extent, so programmers should not store without first copying them.

body is the body of the translator, and is run in the context of the application. *body* may have zero or more declarations as its first forms. It should return either one, two, or three values. The first value is an object which must be `presentation-typep` of *to-type*, and the second value is a presentation type that must be `presentation-subtypep` of *to-type*. The consequences are unspecified if the object is not `presentation-typep` of *to-type* or the type is not `presentation-subtypep` of *to-type*. The first two returned values of *body* are used, in effect, as the returned values for the call to `accept` that established the matching input context.

The third value returned by *body* must either be `nil` or a list of options (as keyword-value pairs) that will be interpreted by `accept`. The only option defined so far is `:echo`, whose value must be either *true* (the default) or *false*. If it is *true*, the object returned by the translator will be "echoed" by `accept`, which will use `presentation-replace-input` to insert the textual representation of the object into the input buffer. If it is *false*, the object will not be echoed.

None of `define-presentation-translator`'s arguments is evaluated.

⇒ `define-presentation-to-command-translator` *name (from-type command-name command-table* `&key` *gesture tester documentation pointer-documentation menu priority echo) arglist* `&body` *body* [*Macro*]

This is similar to `define-presentation-translator`, except that the *to-type* will be derived to be the command named by *command-name* in the command table *command-table*. *command-name* is the name of the command that this translator will translate to.

The *echo* option is a boolean value (the default is *true*) that indicates whether the command line should be echoed when a user invokes the translator.

The other arguments to `define-presentation-to-command-translator` are the same as for `define-presentation-translator`. Note that the tester for command translators is always assumed to be definitive, so there is no `:tester-definitive` option. The default for *pointer-documentation* is the string *command-name* with dash characters replaced by spaces, and each word capitalized (as in `add-command-to-command-table`).

The body of the translator must return a list of the arguments to the command named by *command-name*. *body* is run in the context of the application. The returned value of the body, appended to the command name, are eventually passed to `execute-frame-command`. *body* may have zero or more declarations as its first forms.

None of `define-presentation-to-command-translator`'s arguments is evaluated.

⇒ `define-presentation-action` *name (from-type to-type command-table* `&key` *gesture tester documentation pointer-documentation menu priority) arglist* `&body` *body* [*Macro*]

`define-presentation-action` is similar to `define-presentation-translator`, except that the body of the action is not intended to return a value, but should instead side-effect some sort of application state.

A presentation action does not satisfy a request for input the way an ordinary translator does. Instead, an action is something that happens while waiting for input. After the action has been executed, the program continues to wait for the same input that it was waiting for prior to executing the action.

The other arguments to `define-presentation-action` are the same as for `define-presentation-translator`. Note that the tester for presentation actions is always assumed to be definitive.

None of `define-presentation-action`'s arguments is evaluated.

⇒ `define-drag-and-drop-translator` *name (from-type to-type destination-type command-table* `&key` *gesture tester documentation pointer-documentation menu priority feedback highlighting) arglist* `&body` *body* [*Macro*]

Defines a "drag and drop" (or "direct manipulation") translator named *name* that translates from objects of type *from-type* to objects of type *to-type* when a "from presentation" is "picked up", "dragged" over, and "dropped" on to a "to presentation" having type *destination-type*. *from-type*, *to-type*, and *destination-type* are presentation type specifiers, but must not include any presentation type options. *from-type*, *to-type* and *destination-type* may be presentation type abbreviations.

The interaction style used by these translators is that a user points to a "from presentation" with the pointer, picks it up by pressing a pointer button matching *gesture*, drags the "from presentation" to a "to presentation" by moving the pointer, and then drops the "from presentation" onto the "to presentation". The dropping might be accomplished by either releasing the pointer button or clicking again, depending on the frame manager. When the pointer button is released, the translator whose *destination-type* matches the presentation type of the "to presentation" is chosen. For example, dragging a file to the TrashCan on a Macintosh could be implemented by a drag and drop translator.

While the pointer is being dragged, the function specified by *feedback* is invoked to provide feedback to the user. The function is called with eight arguments: the application frame object, the "from presentation", the stream, the initial x and y positions of the pointer, the current x and y positions of the pointer, and a feedback state (either `:highlight` to draw feedback, or `:unhighlight` to erase it). The feedback function is called to draw some feedback the first time pointer moves, and is then called twice each time the pointer moves thereafter (once to erase the previous feedback, and then to draw the new feedback). It is called a final time to erase the last feedback when the pointer button is released. *feedback* defaults to `frame-drag-and-drop-feedback`.

When the "from presentation" is dragged over any other presentation that has a direct manipulation translator, the function specified by *highlighting* is invoked to highlight that object. The function is called with four arguments: the application frame object, the "to presentation" to be highlighted or unhighlighted, the stream, and a highlighting state (either `:highlight` or `:unhighlight`). *highlighting* defaults to `frame-drag-and-drop-highlighting`.

Note that it is possible for there to be more than one drag and drop translator that applies to the same from-type, to-type, and gesture. In this case, the exact translator that is chosen for use during the dragging phase is unspecified. If these translators have different feedback, highlighting, documentation, or pointer documentation, the exact behavior is unspecified.

The other arguments to `define-drag-and-drop-translator` are the same as for `define-presentation-translator`.

23.7.2 Presentation Translator Functions

⇒ `find-presentation-translators` *from-type to-type command-table* [*Function*]

Returns a list of all of the translators in the command table *command-table* that translate from *from-type* to *to-type*, without taking into account any type parameters or testers. *from-type* and *to-type* are presentation type specifiers, and must not be abbreviations. *frame* must be an application frame.

Implementation note: Because `find-presentation-translators` is called during pointer sensitivity computations (that is, whenever the user mouses the pointer around in any CLIM pane), it should cache its result in order to avoid consing. Therefore, the resulting list of translators should not be modified; the consequences of doing so are unspecified.

Implementation note: The ordering of the list of translators is left unspecified, but implementations may find it convenient to return the list using the ordering specified for `find-applicable-translators`.

⇒ `test-presentation-translator` *translator presentation context-type frame window x y* `&key` *event modifier-state for-menu* [*Function*]

Returns *true* if the translator *translator* applies to the presentation *presentation* in input context type *context-type*, otherwise returns *false*. (There is no *from-type* argument because it is derived from *presentation*.) *x* and *y* are the *x* and *y* positions of the pointer within the window stream *window*.

event and *modifier-state* are a pointer button event and modifier state (see `event-modifier-key-state`), and are compared against the translator's gesture. *event* defaults to `nil`, and *modifier-state* defaults to 0, meaning that no modifier keys are held down. Only one of *event* or *modifier-state* may be supplied; it is unspecified what will happen if both are supplied.

If *for-menu* is *true*, the comparison against *event* and *modifier-state* is not done.

presentation, *context-type*, *frame*, *window*, *x*, *y*, and *event* are passed along to the translator's tester if and when the tester is called.

`test-presentation-translator` is responsible for matching type parameters and calling the translator's tester. Under some circumstances, `test-presentation-translator` may also call the body of the translator to ensure that its value matches *to-type*.

⇒ `find-applicable-translators` *presentation input-context frame window x y* `&key` *event modifier-state for-menu fastp* [*Function*]

Returns a list that describes the translators that definitely apply to the presentation *presentation* in the input context *input-context*. Each element in the returned list is of the form
(translator the-presentation context-type . rest)
where *translator* is a presentation translator, *the-presentation* is the presentation that the translator applies to (and can be different from *presentation* due to nesting of presentations), *context-type* is the context type in which the translator applies, and *rest* is other unspecified data reserved for internal use by CLIM. *translator*, *the-presentation*, and *context-type* can be passed to such functions as `call-presentation-translator` and `document-presentation-translator`.

Since input contexts can be nested, `find-applicable-translators` must iterate over all the contexts in *input-context*. *window*, *x*, and *y* are as for `test-presentation-translator`. *event* and *modifier-state* (which default to `nil` and the current modifier state for *window*, respectively) are used to further restrict the set of applicable translators. (Only one of *event* or *modifier-state* may be supplied; it is unspecified what will happen if both are supplied.)

Presentations can also be nested. The ordering of the translators returned by `find-applicable-`

`translators` is that translators matching inner contexts should precede translators matching outer contexts, and, in the same input context, inner presentations precede outer presentations.

When *for-menu* is non-`nil`, this matches the value of *for-menu* against the presentation's menu specification, and returns only those translators that match. *event* and *modifier-state* are disregarded in this case. *for-menu* defaults to `nil`.

When the boolean *fastp* is *true*, `find-applicable-translators` will simply return *true* if there are any translators. *fastp* defaults to *false*.

When *fastp* is *false*, the list of translators returned by `find-applicable-translators` must be in order of their "desirability", that is, translators having more specific from-types and/or higher priorities must precede translators having less specific from-types and lower priorities.

The rules used for ordering the translators returned by `find-applicable-translators` are as follows (in order):

1. Translators with a higher "high order" priority precede translators with a lower "high order" priority. This allows programmers to set the priority of a translator in such a way that it always precedes all other translators.
2. Translators with a more specific "from type" precede translators with a less specific "from type".
3. Translators with a higher "low order" priority precede translators with a lower "low order" priority. This allows programmers to break ties between translators that translate from the same type.
4. Translators from the current command table precede translators inherited from superior command tables.

Implementation note: `find-applicable-translators` could be implemented by looping over *input-context*, calling `find-presentation-translators` to generate all the translators, and then calling `test-presentation-translator` to filter out the ones that do not apply. The consequences of modifying the returned value are unspecified. Note that the ordering of translators can be done by `find-presentation-translators`, provided that `find-applicable-translators` takes care to preserve this ordering.

Minor issue: *Describe and implement the* `class-nondisjoint-classes` *idea. Be very clear and precise about when the translator body gets run. — SWM*

⇒ `presentation-matches-context-type` *presentation context-type frame window x y* `&key` *event modifier-state* [*Function*]

Returns *true* if there are any translators that translate from the *presentation presentation*'s type to the input context type *context-type*, otherwise returns *false*. (There is no *from-type* argument because it is derived from *presentation*.) *frame*, *window*, *x*, *y*, *event*, and *modifier-state* are as for `test-presentation-translator`.

If there are no applicable translators, `presentation-matches-context-type` will return *false*.

⇒ `call-presentation-translator` *translator presentation context-type frame event window x y* [*Function*]

Calls the function that implements the body of the translator *translator* on the *presentation presentation*'s object, and passes *presentation*, *context-type*, *frame*, *event*, *window*, *x*, and *y* to the body of the translator as well.

The returned values are the same as the values returned by the body of the translator, namely, the translated object and the translated type.

⇒ `document-presentation-translator` *translator presentation context-type frame event window x y* `&key` *(stream* `*standard-output*`*) documentation-type* [*Function*]

Computes the documentation string for the translator *translator* and outputs it to the stream *stream. presentation*, *context-type*, *frame*, *event*, *window*, *x*, and *y* are as for `test-presentation-translator`.

`documentation-type` must be either `:normal` or `:pointer`. If it is `:normal`, the usual translator documentation function is called. If it is `:pointer`, the translator's pointer documentation is called.

⇒ `call-presentation-menu` *presentation input-context frame window x y* `&key` *for-menu label* [*Function*]

Finds all the applicable translators for the *presentation presentation* in the input context *input-context*, creates a menu that contains all of the translators, and pops up the menu from which the user can choose a translator. After the translator is chosen, it is called with the arguments supplied to `call-presentation-menu` and the matching input context that was established by `with-input-context` is terminated.

window, *x*, *y*, and *event* are as for `find-applicable-translators`. *for-menu*, which defaults to `t`, is used to decide which of the applicable translators will go into the menu; only those translators whose `:menu` option matches *menu* will be included.

label is either a string to use as a label for the menu, or is `nil` (the default), meaning the menu will not be labelled.

23.7.3 Finding Applicable Presentations

⇒ `find-innermost-applicable-presentation` *input-context window x y* `&key` *frame modifier-state event* [*Function*]

Given an input context *input-context*, an output recording window stream *window*, *x* and *y* positions *x* and *y*, returns the innermost presentation whose sensitivity region contains *x* and *y* that matches the innermost input context, using the translator matching algorithm described below. If there is no such presentation, this function will return `nil`.

event and *modifier-state* are a pointer button event and modifier state (see `event-modifier-key-state`). *event* defaults to `nil`, and *modifier-state* defaults to the current modifier state for *window*. Only one of *event* or *modifier-state* may be supplied; it is unspecified what will happen if both are supplied.

frame defaults to the current frame, `*application-frame*`.

The default method for `frame-find-innermost-applicable-presentation` will call this function.

⇒ `throw-highlighted-presentation` *presentation input-context button-press-event* [*Function*]

Given a *presentation presentation*, input context *input-context*, and a button press event (which contains the window, pointer, x and y position of the pointer within the window, the button pressed, and the modifier state), find the translator that matches the innermost presentation in the innermost input context, then call the translator to produce an object and a presentation type. Finally, the matching input context that was established by `with-input-context` will be terminated.

Note that it is possible that more than one translator having the same gesture may be applicable to *presentation* in the specified input context. In this case, the translator having the highest priority will be chosen. If there is more than one having the same priority, it is unspecified what translator will be chosen.

⇒ `highlight-applicable-presentation` *frame stream input-context* `&optional` *prefer-pointer-window* [*Function*]

This is the core of the "input wait" handler used by `with-input-context` on behalf of the *application frame frame*. It is responsible for locating the innermost applicable presentation on *stream* in the input context *input-context*, unhighlighting presentations that are not applicable, and highlighting the presentation that is applicable. Typically on entry to `highlight-applicable-presentation`, *input-context* will be the value of `*input-context*` and *frame* will be the value of `*application-frame*`.

If *prefer-pointer-window* is *true* (the default), CLIM will highlight the applicable presentation on the same window that the pointer is located over. Otherwise, CLIM will highlight an applicable presentation on *stream*.

Implementation note: This will probably use `frame-find-innermost-applicable-presentation-at-position` to locate the innermost presentation, and `unhighlight-highlighted-presentation` and `set-highlighted-presentation` to unhighlight and highlight presentations.

⇒ `set-highlighted-presentation` *stream presentation* `&optional` *prefer-pointer-window* [*Function*]

Highlights the *presentation presentation* on *stream*. This must call `highlight-presentation` methods if that is appropriate.

prefer-pointer-window is as for `highlight-applicable-presentation`.

⇒ `unhighlight-highlighted-presentation` *stream* `&optional` *prefer-pointer-window* [*Function*]

Unhighlights any highlighted presentations on *stream*.

prefer-pointer-window is as for `highlight-applicable-presentation`.

23.7.4 Translator Applicability

The top-level "input wait", which is what you are in when inside of a `with-input-context`, is responsible for determining what translators are applicable to which presentations in a given input context. This loop both provides feedback in the form of highlighting sensitive presentation, and is responsible for calling the applicable translator when the user presses a pointer button.

Implementation note: `with-input-context` uses `frame-find-innermost-applicable-presentation-at-position` (via `highlight-applicable-presentation`) as its "input wait" handler, and `frame-input-context-button-press-handler` as its button press "event handler".

Given a presentation, an input context established by `with-input-context`, and an event corresponding to a user gesture, translator matching proceeds as follows.

The set of candidate translators is initially those translators accessible in the command table in use by the current application. A translator is said to "match" if all of the following are true (in this order):

1. The presentation's type is `presentation-subtypep` of the translator's *from-type*, ignoring type parameters.
2. The translator's *to-type* is `presentation-subtypep` of the input context type, ignoring type parameters.
3. The translator's gesture is either `t`, or matches the event corresponding to the user's gesture.
4. If there are parameters in the *from-type*, the presentation's object must be `presentation-typep` of the *from-type*.
5. The translator's tester returned *true*. If there is no tester, the translator behaves as though there is a tester that always returns *true*.
6. If there are parameters in the input context type and the tester is not declared to be definitive, the value returned by body of the translator must be `presentation-typep` of the context type.

Note that the type parameters from the presentation's type have no effect on translator lookup.

`find-presentation-translator` is responsible for the first two steps of the matching algorithm, and `test-presentation-translator` is responsible for the remaining steps.

When a single translator is being chosen (such as is done by `throw-highlighted-presentation`), it is possible that more than one translator having the same gesture may be applicable to the presentation in the specified input context. In this case, the translator having the highest priority will be chosen. If there is more than one having the same priority, it is unspecified what translator will be chosen.

The matching algorithm is somewhat more complicated in face of nested presentations and nested input contexts. In this case, the applicable presentation is the *smallest* presentation that matches the *innermost* input context.

Sometimes there may be nested presentations that have exactly the same bounding rectangle. In this case, it is not possible for a user to unambiguously point to just one of the nested presentations. Therefore, when CLIM has located the innermost applicable presentation in the innermost input context, it must then search for outer presentations having exactly the same bounding rectangle, checking to see if there are any applicable translators for those presentations. If there are multiple applicable translators, the one having the highest priority is chosen. `find-applicable-translators`, `call-presentation-menu`, `throw-highlighted-presentation`, and the computation of pointer documentation must all take this situation into account.

The translators are searched in the order that they are returned by `find-presentation-translators`. The rules for the ordering of the translators are described under that function.

23.8 Standard Presentation Types

The following sections document the presentation types supplied by CLIM. Any presentation type with the same name as a Common Lisp type accepts the same parameters as the Common Lisp type (and additional parameters in a few cases).

23.8.1 Basic Presentation Types

⇒ `t` *[Presentation Type]*

The supertype of all other presentation types.

⇒ `nil` *[Presentation Type]*

The subtype of all other presentation types. This has no printed representation, and it useful only in writing "context independent" translators, that is, translators whose *to-type* is `nil`.

⇒ `null` *[Presentation Type]*

The type that represents "nothing". The single object associated with this type is `nil`, and its printed representation is "None".

⇒ `boolean` *[Presentation Type]*

The type that represents *true* or *false*. The printed representation is "Yes" or "No", respectively.

⇒ `symbol` [*Presentation Type*]

The type that represents a symbol.

⇒ `keyword` [*Presentation Type*]

The type that represents a symbol in the keyword package. It is a subtype of `symbol`.

⇒ `blank-area` [*Presentation Type*]

The type that represents all the places in a window where there is no presentation that is applicable in the current input context. CLIM provides a single "null presentation" as the object associated with this type.

⇒ `*null-presentation*` [*Constant*]

The null presentation, which occupies all parts of a window in which there are no applicable presentations. This will have a presentation type of `blank-area`.

23.8.2 Numeric Presentation Types

⇒ `number` [*Presentation Type*]

The type that represents a general number. It is the supertype of all the number types.

⇒ `complex &optional` *type* [*Presentation Type*]

The type that represents a complex number. It is a subtype of `number`.

The components of the complex number are of type *type*, which must be `real` or a subtype of `real`.

⇒ `real &optional` *low high* [*Presentation Type*]

The type that represents either a ratio, an integer, or a floating point number between *low* and *high*. *low* and *high* can be inclusive or exclusive, as in Common Lisp type specifiers. Options to this type are *base* (default 10) and *radix* (default `nil`). `real` is a subtype of `number`.

⇒ `rational &optional` *low high* [*Presentation Type*]

The type that represents either a ratio or an integer between *low* and *high*. Options to this type are *base* and *radix*. `rational` is a subtype of `real`.

⇒ `integer &optional` *low high* [*Presentation Type*]

The type that represents an integer between *low* and *high*. Options to this type are *base* and *radix*. `integer` is a subtype of `rational`.

⇒ `ratio &optional` *low high* [*Presentation Type*]

The type that represents a ratio between *low* and *high*. Options to this type are *base* and *radix*. `ratio` is a subtype of `rational`.

⇒ `float &optional` *low high* [*Presentation Type*]

The type that represents a floating point number between *low* and *high*. `float` is a subtype of `number`.

23.8.3 Character and String Presentation Types

⇒ `character` [*Presentation Type*]

The type that represents a character object.

⇒ `string &optional` *length* [*Presentation Type*]

The type that represents a string. If *length* is supplied, the string must contain exactly that many characters.

23.8.4 Pathname Presentation Type

⇒ `pathname` [*Presentation Type*]

The type that represents a pathname. The options are *default-version*, which defaults to `:newest`, *default-type*, which defaults to `nil`, and *merge-default*, which defaults to *true*. If *merge-default* is *false*, `accept` returns the exact pathname that was entered, otherwise `accept` merges against the default and *default-version*. If no default is supplied, it defaults to `*default-pathname-defaults*`. The `pathname` type should have a default preprocessor that merges the options into the default.

23.8.5 "One-of" and "Some-of" Presentation Types

⇒ `completion` *sequence* `&key` *test value-key* [*Presentation Type*]

The type that selects one from a finite set of possibilities, with "completion" of partial inputs. The member types below, `token-or-type`, and `null-or-type` are implemented in terms of the `completion` type.

sequence is a list or vector whose elements are the possibilities. Each possibility has a printed representation, called its name, and an internal representation, called its value. `accept` reads a name and returns a value. `present` is given a value and outputs a name.

test is a function that compares two values for equality. The default is `eql`.

value-key is a function that returns a value given an element of *sequence*. The default is `identity`.

The following presentation type options are available:

name-key is a function that returns a name, as a string, given an element of *sequence*. The default is a function that behaves as follows:

string	⇒	the string
null	⇒	`"NIL"`
cons	⇒	`string` of the `car`
symbol	⇒	`string-capitalize` of its name
otherwise	⇒	`princ-to-string` of it

documentation-key is a function that returns either `nil` or a descriptive string, given an element of *sequence*. The default always returns `nil`.

test, *value-key*, *name-key*, and *documentation-key* must have indefinite extent.

partial-completers is a possibly-empty list of characters that delimit portions of a name that can be completed separately. The default is a list of one character, `#\Space`.

⇒ `member &rest` *elements* [*Presentation Type Abbreviation*]

The type that specifies one of *elements*. The options are the same as for `completion`.

⇒ `member-sequence` *sequence* `&key` *test* [*Presentation Type Abbreviation*]

Like `member`, except that the set of possibilities is the sequence *sequence*. The parameter *test* and the options are the same as for `completion`.

⇒ `member-alist` *alist* `&key` *test* [*Presentation Type Abbreviation*]

Like `member`, except that the set of possibilities is the alist *alist*. Each element of *alist* is either an atom as in `member-sequence` or a list whose `car` is the name of that possibility and whose `cdr` is one of the following:

- The value (which must not be a cons)
- A list of one element, the value
- A property list that can contain the following properties:
  - `:value`—the value
  - `:documentation`—a descriptive string

The *test* parameter and the options are the same as for `completion` except that *value-key* and *documentation-key* default to functions that support the specified alist format.

⇒ `subset-completion` *sequence* `&key` *test value-key* [*Presentation Type*]

The type that selects one or more from a finite set of possibilities, with "completion" of partial inputs. The parameters and options are the same as for `completion`, plus the additional options *separator* and *echo-space*, which are as for the `sequence` type. The subset types below are implemented in terms of the `subset-completion` type.

⇒ `subset &rest` *elements* [*Presentation Type Abbreviation*]

The type that specifies a subset of *elements*. Values of this type are lists of zero or more values chosen from the possibilities in *elements*. The printed representation is the names of the elements separated by commas. The options are the same as for `completion`.

⇒ `subset-sequence` *sequence* `&key` *test* [*Presentation Type Abbreviation*]

Like `subset`, except that the set of possibilities is the sequence *sequence*. The parameter *test* and the options are the same as for `completion`.

⇒ `subset-alist` *alist* `&key` *test* [*Presentation Type Abbreviation*]

Like `subset`, except that the set of possibilities, the parameters, and the options are as for `member-alist`.

23.8.6 Sequence Presentation Types

⇒ `sequence` *type* [*Presentation Type*]

The type that represents a sequence of elements of type *type*. *type* can be a presentation type abbreviation. The printed representation of a `sequence` type is the elements separated by commas. It is unspecified whether `accept` returns a list or a vector.

The options to this type are *separator* and *echo-space*. *separator* is used to specify a character that will act as the separator between elements of the sequence; the default is the comma character `#\,`. *echo-space* must be *true* or *false*; when it is *true* (the default) a space will be automatically inserted into the input buffer when the user types a separator character.

⇒ `sequence-enumerated &rest` *types* [*Presentation Type*]

`sequence-enumerated` is like `sequence`, except that the type of each element in the sequence is individually specified. The elements of *types* can be presentation type abbreviations. It is unspecified whether `accept` returns a list or a vector.

The options to this type are *separator* and *echo-space*, which are as for the `sequence` type.

23.8.7 "Meta" Presentation Types

⇒ `or &rest` *types* [*Presentation Type*]

The type that is used to specify one of several types, for example, `(or (member :all :none) integer)`. The elements of *types* can be presentation type abbreviations. `accept` returns one of the possible types as its second value, not the original `or` presentation type specifier.

The `accept` method for `or` could be implemented by iteratively calling `accept` on each of the presentation types in *types*. It would establish a condition handler for `parse-error`, call `accept` on one of the types and return the result if no condition was signalled. If a `parse-error` is signalled, the `accept` method for `or` would call `accept` on the next type. When there are no more types, the `accept` method for `or` would itself signal a `parse-error`.

⇒ `and &rest` *types* [*Presentation Type*]

The type that is used for "multiple inheritance". `and` is frequently used in conjunction with `satisfies`, for example, `(and integer (satisfies oddp))`. The elements of *types* can be presentation type abbreviations.

The `and` type has special syntax that supports the two "predicates", `satisfies` and `not`. `satisfies` and `not` cannot stand alone as presentation types and cannot be first in *types*. `not` can surround either `satisfies` or a presentation type.

The first type in *types* is the type whose methods will be used during calls to `accept` and `present`.

23.8.8 Compound Presentation Types

⇒ `token-or-type` *tokens type* [*Presentation Type Abbreviation*]

A compound type that is used to select one of a set of special tokens, or an object of type *type*. *tokens* is anything that can be used as the *sequence* parameter to `member-alist`; typically it is a list of symbols.

⇒ `null-or-type` *type* [*Presentation Type Abbreviation*]

A compound type that is used to select `nil`, whose printed representation is the special token "None", or an object of type *type*.

⇒ `type-or-string` *type* [*Presentation Type Abbreviation*]

A compound type that is used to select an object of type *type* or an arbitrary string, for example, `(type-or-string integer)`. Any input that `accept` cannot parse as the representation of an object of type *type* is returned as a string.

23.8.9 Lisp Expression Presentation Types

⇒ `expression` [*Presentation Type*]

The type used to represent any Lisp object. The standard `print` and `read` functions produce and accept the textual view of this type.

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

⇒ `form` *[Presentation Type]*

The type used to represent a Lisp form. This is a subtype of `expression` and is equivalent except that some presentation translators produce `quote` forms.

Chapter 24

Input Editing and Completion Facilities

CLIM provides number of facilities to assist in writing presentation type parser functions, such as an interactive input editor and some "completion" facilities.

24.1 The Input Editor

An input editing stream "encapsulates" an interactive stream, that is, most operations are handled by the encapsulated interactive stream, but some operations are handled directly by the input editing stream itself. (See Appendix C for a discussion of encapsulating streams.)

An input editing stream will have the following components:

- The encapsulated interactive stream.
- A buffer with a fill pointer, which we shall refer to as FP. The buffer contains all of the user's input, and FP is the length of that input.
- An insertion pointer, which we shall refer to as IP. The insertion pointer is the point in the buffer at which the "editing cursor" is.
- A scan pointer, which we shall refer to as SP. The scan pointer is the point in the buffer from which CLIM will get the next input gesture object (in the sense of `read-gesture`).
- A "rescan queued" flag indicating that the programmer (or CLIM) requested that a "rescan" operation should take place before the next gesture is read from the user.
- A "rescan in progress" flag that indicates that CLIM is rescanning the user's input, rather than reading freshly supplied gestures from the user.

The input editing stream may also have other components to store internal state, such as a slot to accumulate a numeric argument or remember the most recently used presentation history, and so forth. These other components are explicitly left unspecified.

The high level description of the operation of the input editor is that it reads either "real" gestures from the user (such as characters from the keyboard or pointer button events) or input editing commands. The input editing commands can modify the state of the input buffer. When such modifications take place, it is necessary to "rescan" the input buffer, that is, reset the scan pointer SP to its original state and reparse the contents of the input editor buffer before reading any other gestures from the user. While this rescanning operation is taking place, the "rescan in progress" flag is set to *true*. The relationship $SP \leq IP \leq FP$ always holds.

The overall control structure of the input editor is:

```
(catch 'rescan                  ;thrown to when a rescan is invoked
  (reset-scan-pointer stream)   ;sets STREAM-RESCANNING-P to T
  (loop
     (funcall continuation stream)))
```

where *stream* is the input editing stream and *continuation* is the code supplied by the programmer, and typically contains calls to such functions as `accept` and `read-token` (which will eventually call `stream-read-gesture`). When a rescan operation is invoked, it has the effect of throwing to the `rescan` tag in the example above. The loop is terminated when an activation gesture is seen, and at that point the values produced by *continuation* are returned as values from the input editor.

The important point is that functions such as `accept`, `read-gesture`, and `unread-gesture` read (or restore) the next gesture object from the buffer at the position pointed to by the scan pointer SP. However, insertion and input editing commands take place at the position pointed to by IP. The purpose of the rescanning operation is to eventually ensure that all the input gestures issued by the user (typed characters, pointer button presses, and so forth) have been read by CLIM. During input editing, the input editor should maintain some sort of visible cursor to remind the user of the position of IP.

The overall structure of `stream-read-gesture` on an input editing stream is:

```
(progn
  (rescan-if-necessary stream)
  (loop
    ;; If SP is less than FP
    ;;   Then get the next gesture from the input editor buffer at SP
    ;;   and increment SP
    ;;   Else read the next gesture from the encapsulated stream
    ;;   and insert it into the buffer at IP
    ;; Set the "rescan in progress" flag to false
    ;; Call STREAM-PROCESS-GESTURE on the gesture
    ;;   If it was a "real" gesture
    ;;     Then exit with the gesture as the result
    ;;     Else it was an input editing command (which has already been
    ;;     processed), so continue looping
    ))
```

When a new gesture object is inserted into the input editor buffer, it is inserted at the insertion pointer IP. If $IP = FP$, this is accomplished by a `vector-push-extend`-like operation on the input buffer and FP, and then incrementing IP. If $IP < FP$, CLIM must first "make room" for the new gesture in the input buffer, then insert the gesture at IP, then increment both IP and FP.

When the user requests an input editor motion command, only the insertion pointer IP is affected. Motion commands do not need to request a rescan operation.

When the user requests an input editor deletion command, the sequence of gesture objects at IP are removed, and IP and FP must be modified to reflect the new state of the input buffer. Deletion commands (and other commands that modify the input buffer) must arrange for a rescan to occur when they are done modifying the buffer, either by calling `queue-rescan` or `immediate-rescan`.

CLIM implementations are free to put special objects in the input editor buffer, such as "noise strings" and "accept results". A "noise string" is used to represent some sort of in-line prompt and is never seen as input; the `prompt-for-accept` method may insert a noise string into the input buffer. An "accept result" is an object in the input buffer that is used to represent some object that was inserted into the input buffer (typically via a pointer gesture) that has no readable representation (in the Lisp sense); `presentation-replace-input` may create accept results. Noise strings are skipped over by input editing commands, and accept results are treated as a single gesture.

⇒ `interactive-stream-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is an interactive stream, that is, a bidirectional stream intended for user interactions. Otherwise it returns *false*. This is exactly the same function as in X3J13 Common Lisp, except that in CLIM it is a generic function.

The input editor need only be fully implemented for interactive streams.

⇒ `input-editing-stream` [*Protocol Class*]

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

⇒ `input-editing-stream-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is an *input editing stream* (that is, a stream of the sort created by a call to `with-input-editing`), otherwise returns *false*.

⇒ `standard-input-editing-stream` [*Class*]

The instantiable class that implements CLIM's standard input editor. This is the class of stream created by calling `with-input-editing`.

Members of this class are mutable.

⇒ `with-input-editing` *(*`&optional` *stream* `&key` *input-sensitizer initial-contents class)* `&body` *body* [*Macro*]

Establishes a context in which the user can edit the input typed in on the interactive stream *stream*. *body* is then executed in this context, and the values returned by *body* are returned as the values of `with-input-editing`. *body* may have zero or more declarations as its first forms.

The *stream* argument is not evaluated, and must be a symbol that is bound to an input stream. If *stream* is `t` (the default), `*standard-input*` is used. If *stream* is a stream that is not an interactive stream, then `with-input-editing` is equivalent to `progn`.

input-sensitizer, if supplied, is a function of two arguments, a stream and a continuation function; the function has dynamic extent. The continuation, supplied by CLIM, is responsible for displaying output corresponding to the user's input on the stream. The *input-sensitizer* function will typically call `with-output-as-presentation` in order to make the output produced by the continuation sensitive.

If *initial-contents* is supplied, it must be either a string or a list of two elements, an object and a presentation type. If it is a string, the string will be inserted into the input buffer using `replace-input`. If it is a list, the printed representation of the object will be inserted into the input buffer using `presentation-replace-input`.

⇒ `with-input-editor-typeout` *(*`&optional` *stream* `&key` *erase)* `&body` *body* [*Macro*]

Establishes a context inside of `with-input-editing` in which output can be done by *body* to the input editing stream *stream*. If *erase* is *true*, the area underneath the typeout will be erased before the typeout is done. `with-input-editor-typeout` should call `fresh-line` before and after evaluating the body. *body* may have zero or more declarations as its first forms.

The *stream* argument is not evaluated, and must be a symbol that is bound to a stream. If *stream* is `t` (the default), `*standard-input*` is used. If *stream* is a stream that is not an input editing stream, then `with-input-editor-typeout` is equivalent to calling `fresh-line`, evaluating the body, and then calling `fresh-line` again.

⇒ `input-editor-format` *stream format-string* `&rest` *format-args* [*Generic Function*]

This function is like `format`, except that it is intended to be called on input editing streams. It arranges to insert "noise strings" in the input editor's input buffer. Programmers can use this to display in-line prompts in `accept` methods.

If *stream* is a stream that is not an input editing stream, then `input-editor-format` is equivalent to `format`.

24.1.1 The Input Editing Stream Protocol

Input editing streams obey both the extended input and extended output stream protocols, and must support the generic functions that comprise those protocols. For the most part, this will simply entail "trampolining" those operations to the encapsulated interactive stream. However, some generic functions as `stream-read-gesture` and `stream-unread-gesture` will need methods that observe the use of the input editor's scan pointer.

Input editing streams will typically also implement methods for `prompt-for-accept` (in order to provide in-line prompting that interacts correctly with input editing) and `stream-accept` (in order to cause `accept` to obey the scan pointer).

The following generic functions comprise the remainder of the input editing protocol, and must be implemented for all classes that inherit from `input-editing-stream`.

⇒ `stream-input-buffer` *(stream* `input-editing-stream`*)* [*Method*]

Returns the input buffer (that is, the string being edited) associated with the *input editing stream stream*. This must be an unspecialized vector with a fill pointer. The fill pointer of the vector points past the last gesture object in the buffer. During input editing, this buffer is side-effected. The consequences of modifying the input buffer by means other than the specified API (such as `replace-input`) are unspecified.

⇒ `stream-insertion-pointer` *stream* [*Generic Function*]

Returns an integer corresponding to the current input position in the *input editing stream stream*'s buffer, that is, the point in the buffer at which the next user input gesture will be inserted. The insertion pointer will always be less than `(fill-pointer (stream-input-buffer` *stream*`))`. The insertion pointer can also be thought of as an editing cursor.

⇒ `(setf stream-insertion-pointer)` *pointer stream* [*Generic Function*]

Changes the input position of the *input editing stream stream* to *pointer*. *pointer* is an integer, and must be less than `(fill-pointer (stream-input-buffer` *stream*`))`.

⇒ `stream-scan-pointer` *stream* [*Generic Function*]

Returns an integer corresponding to the current scan pointer in the *input editing stream stream*'s buffer, that is, the point in the buffer at which calls to `accept` have stopped parsing input. The scan pointer will always be less than or equal to `(stream-insertion-pointer` *stream*`)`.

⇒ `(setf stream-scan-pointer)` *pointer stream* [*Generic Function*]

Changes the scan pointer of the *input editing stream stream* to *pointer*. *pointer* is an integer, and must be less than or equal to `(stream-insertion-pointer` *stream*`)`.

⇒ `stream-rescanning-p` *stream* [*Generic Function*]

Returns the state of the *input editing stream stream*'s "rescan in progress" flag, which is *true* if *stream* is performing a rescan operation, otherwise it is *false*. All extended input streams must implement a method for this, but non-input editing streams will always returns *false*.

⇒ `reset-scan-pointer` *stream* `&optional` *(scan-pointer* `0`*)* [*Generic Function*]

Sets the *input editing stream stream*'s scan pointer to *scan-pointer*, and sets the state of `stream-rescanning-p` to *true*.

⇒ `immediate-rescan` *stream* [*Generic Function*]

Invokes a rescan operation immediately by "throwing" out to the most recent invocation of `with-input-editing`.

⇒ `queue-rescan` *stream* [*Generic Function*]

Indicates that a rescan operation on the *input editing stream stream* should take place after the next non-input editing gesture is read by setting the "rescan queued" flag to *true*.

⇒ `rescan-if-necessary` *stream* `&optional` *inhibit-activation* [*Generic Function*]

Invokes a rescan operation on the *input editing stream stream* if `queue-rescan` was called on the same stream and no intervening rescan operation has taken place. Resets the state of the "rescan queued" flag to *false*.

If *inhibit-activation* is *false*, the input line will not be activated even if there is an activation character in it.

⇒ `erase-input-buffer` *stream* `&optional` *(start-position* `0`*)* [*Generic Function*]

Erases the part of the display that corresponds to the input editor's buffer starting at the position *start-position*.

⇒ `redraw-input-buffer` *stream* `&optional` *(start-position* `0`*)* [*Generic Function*]

Displays the input editor's buffer starting at the position *start-position* on the interactive stream that is encapsulated by the *input editing stream stream*.

⇒ `stream-process-gesture` *stream gesture type* [*Generic Function*]

If *gesture* is an input editing command, `stream-process-gesture` performs the input editing operation on the *input editing stream stream* and returns `nil`. Otherwise, it returns the two values *gesture* and *type*.

⇒ `stream-read-gesture` *(stream* `standard-input-editing-stream`*)* `&key` [*Method*]

Reads and returns a gesture from the user on the *input editing stream stream*.

The `stream-read-gesture` method must call `stream-process-gesture`, which will either return a "real" gesture (such as a typed character, a pointer gesture, or a timeout) or will return `nil` (indicating that some sort of input editing operation was performed). `stream-read-gesture` must only return when a real gesture was been read; if an input editing operation was performed, `stream-read-gesture` will loop until a "real" gesture is typed by the user.

⇒ `stream-unread-gesture` *(stream* `standard-input-editing-stream`*) gesture* [*Method*]

Inserts the gesture *gesture* back into the input editor's buffer, maintaining the scan pointer.

24.1.2 Suggestions for Input Editing Commands

An implementation of the input editor should provide a set of generally useful input editing commands. The exact set of these commands is unspecified, and the key bindings for these commands may vary from platform to platform. The following is a suggested minimum set of input editing commands and key bindings, taken roughly from EMACS.

	Suggested
Input editor command	*key binding*
Forward character	`control-F`
Forward word	`meta-F`
Backward character	`control-B`
Backward word	`meta-B`
Beginning of line	`control-A`
End of line	`control-E`
Next line	`control-N`
Previous line	`control-P`
Beginning of buffer	`meta-<`
End of buffer	`meta-<`
Delete next character	`control-D`
Delete next word	`meta-D`
Delete previous character	`Rubout`
Delete previous word	`m-Rubout`
Kill to end of line	`control-K`
Clear input buffer	*varies*
Insert new line	`control-O`
Transpose adjacent characters	`control-T`
Transpose adjacent words	`meta-T`
Yank from kill ring	`control-Y`
Yank from presentation history	`control-meta-Y`
Yank next item	`meta-Y`
Scroll output history forward	`control-V`
Scroll output history backward	`meta-V`

An implementation of the input may also support "numeric arguments" (such as `control-0`, `control-1`, `meta-0`, and so forth) that modify the behavior of the input editing commands. For instance, the motion and deletion commands should be repeated as many times as specified by the numeric argument. Furthermore, the accumulated numeric argument should be passed to the command processor in such a way that `substitute-numeric-argument-marker` can be used to insert the numeric argument into a command that was read via a keystroke accelerator.

⇒ `add-input-editor-command` *gestures function* [*Function*]

Adds an input editing command that causes *function* to be executed when the specified gesture(s) are typed by the user. *gestures* is either a single gesture name, or a list of gesture names. When *gestures* is a sequence of gesture names, the function is executed only after all of the gestures are typed in order with no intervening gestures. (This is used to implement "prefixed" commands, such as the `control-X control-F` command one might fix in EMACS.)

24.2 Activation and Delimiter Gestures

Activation gestures terminate an input "sentence", such as a command or anything else being read by `accept`. When an activation gesture is entered by the user, CLIM will cease reading input and "execute" the input that has been entered.

Delimiter gestures terminate an input "word", such as a recursive call to `accept`.

⇒ `*activation-gestures*` [*Variable*]

The set of currently active activation gestures. The global value of this must be `nil`. The exact format of `*activation-gestures*` is unspecified. `*activation-gestures*` and the elements in it may have dynamic extent.

⇒ `*standard-activation-gestures*` [*Variable*]

The default set of activation gestures. The exact set of standard activation is unspecified, but must include the gesture that corresponds to the `#\Newline` character.

⇒ `with-activation-gestures` *(gestures* `&key` *override)* `&body` *body* [*Macro*]

Specifies a list of gestures that terminate input during the execution of *body*. *body* may have zero or more declarations as its first forms. *gestures* must be either a single gesture name or a form that evaluates to a list of gesture names.

If the boolean *override* is *true*, then *gestures* will override the current activation gestures. If it is *false* (the default), then *gestures* will be added to the existing set of activation gestures. `with-activation-gestures` must bind `*activation-gestures*` to the new set of activation gestures.

See also the `:activation-gestures` and `:additional-activation-gestures` options to `accept`.

⇒ `activation-gesture-p` *gesture* [*Function*]

Returns *true* if the gesture object *gesture* is an activation gesture, otherwise returns *false*.

⇒ `*delimiter-gestures*` [*Variable*]

The set of currently active delimiter gestures. The global value of this must be `nil`. The exact format of `*delimiter-gestures*` is unspecified. `*delimiter-gestures*` and the elements in it may have dynamic extent.

⇒ `with-delimiter-gestures` *(gestures* `&key` *override)* `&body` *body* [*Macro*]

Specifies a list of gestures that terminate an individual token, but not the entire input, during the execution of *body*. *body* may have zero or more declarations as its first forms. *gestures* must be either a single gesture name or a form that evaluates to a list of gesture names.

If the boolean *override* is *true*, then *gestures* will override the current delimiter gestures. If it is *false* (the default), then *gestures* will be added to the existing set of delimiter gestures. `with-delimiter-gestures` must bind `*delimiter-gestures*` to the new set of delimiter gestures.

See also the `:delimiter-gestures` and `:additional-delimiter-gestures` options to `accept`.

⇒ `delimiter-gesture-p` *gesture* [*Function*]

Returns *true* if the gesture object *gesture* is a delimiter gesture, otherwise returns *false*.

24.3 Signalling Errors Inside `present` Methods

⇒ `simple-parse-error` [*Error Condition*]

The error that is signalled by `simple-parse-error`. This is a subclass of `parse-error`.

This condition handles two initargs, `:format-string` and `:format-arguments`, which are used to specify a control string and arguments for a call to `format`.

⇒ `simple-parse-error` *format-string* `&rest` *format-arguments* [*Function*]

Signals a `simple-parse-error` error while parsing an input token. Does not return. *format-string* and *format-args* are as for `format`.

⇒ `input-not-of-required-type` [*Error Condition*]

The error that is signalled by `input-not-of-required-type`. This is a subclass of `parse-error`.

This condition handles two initargs, `:string` and `:type`, which specify a string to be used in an error message and the expected presentation type.

⇒ `input-not-of-required-type` *object type* [*Function*]

Reports that input does not satisfy the specified type by signalling an `input-not-of-required-type` error. *object* is a parsed object or an unparsed token (a string). *type* is a presentation type specifier. Does not return.

24.4 Reading and Writing of Tokens

⇒ `replace-input` *stream new-input* `&key` *start end buffer-start rescan* [*Generic Function*]

Replaces the part of the *input editing stream stream*'s input buffer that extends from *buffer-start* to its scan pointer with the string *new-input*. *buffer-start* defaults to the current input position of *stream*. *start* and *end* can be supplied to specify a subsequence of *new-input*; *start* defaults to 0 and *end* defaults to the length of *new-input*.

`replace-input` must queue a rescan by calling `queue-rescan` if the new input does not match the old input, or *rescan* is *true*.

The returned value is the position in the input buffer.

All input editing streams must implement a method for this function.

⇒ `presentation-replace-input` *stream object type view* `&key` *buffer-start rescan query-identifier for-context-type* [*Generic Function*]

Like `replace-input`, except that the new input to insert into the input buffer is gotten by presenting *object* with the presentation type *type* and view *view*. *buffer-start* and *rescan* are as for `replace-input`, and *query-identifier* and *for-context-type* as as for `present`.

All input editing streams must implement a method for this function. Typically, this will be implemented by calling `present-to-string` on *object*, *type*, *view*, and *for-context-type*, and then calling `replace-input` on the resulting string.

If the object does not have a readable representation (in the Lisp sense), `presentation-replace-input` may create an "accept result" to represent the object, and insert that into the input buffer. For the purposes of input editing, "accept results" must be treated as a single input gesture.

⇒ `read-token` *stream* `&key` *input-wait-handler pointer-button-press-handler click-only* [*Function*]

Reads characters from the *interactive stream stream* until it encounters a delimiter or activation gesture, or a pointer gesture. Returns the accumulated string that was delimited by the delimiter or activation gesture, leaving the delimiter unread.

If the first character of typed input is a quotation mark (`#\"`), then `read-token` will ignore delimiter gestures until until another quotation mark is seen. When the closing quotation mark is seen, `read-token` will proceed as above.

If the boolean *click-only* is *true*, then no keyboard input is allowed. In this case `read-token` will simply ignore any typed characters.

input-wait-handler and *pointer-button-press-handler* are as for `stream-read-gesture`.

⇒ `write-token` *token stream* `&key` *acceptably* [*Function*]

`write-token` is the opposite of `read-token` given the string *token*, it writes it to the *interactive stream stream*. If *acceptably* is *true* and there are any characters in the token that are delimiter gestures (see the macro `with-delimiter-gestures`), then `write-token` will surround the token with quotation marks (`#\"`).

Typically, `present` methods will use `write-token` instead of `write-string`.

24.5 Completion

CLIM provides a *completion* facility that completes a string provided by a user against some set of possible completions (which are themselves strings). Each completion is associated with some Lisp object. CLIM implementations are encouraged to provide "chunkwise" completion, that is, if the user input consists of several tokens separated by "partial delimiters", CLIM should complete each token separately against the set of possibilities.

⇒ `*completion-gestures*` [*Variable*]

A list of the gesture names that cause `complete-input` to complete the user's input as fully as possible. The exact global contents of this list is unspecified, but must include the `:complete` gesture name.

⇒ `*help-gestures*` [*Variable*]

A list of the gesture names that cause `accept` and `complete-input` to display a (possibly input context-sensitive) help message, and for some presentation types a list of possibilities as well. The exact global contents of this list is unspecified, but must include the `:help` gesture name.

⇒ `*possibilities-gestures*` [*Variable*]

A list of the gesture names that cause `complete-input` to display a (possibly input context-sensitive) help message and a list of possibilities. The exact global contents of this list is unspecified, but must include the `:possibilities` gesture name.

⇒ `complete-input` *stream function* `&key` *partial-completers allow-any-input possibility-printer (help-displays-possibilities t)* [*Function*]

Reads input from the user from the *input editing stream stream*, completing over a set of possibilities. `complete-input` is only required to work on input editing streams, but implementations may extend it to work on interactive streams as well.

function is a function of two arguments. It is called to generate the completion possibilities that match the user's input; it has dynamic extent. Usually, programmers will pass either `complete-from-possibilities` or `complete-from-generator` as the value of *function*. Its first argument is a string containing the user's input "so far". Its second argument is the completion mode, one of the following:

- `:complete-limited`—the function must complete the input up to the next partial delimiter. This is the mode used when the user types one of the partial completers.
- `:complete-maximal`—the function must complete the input as much as possible. This is the mode used when the user issues a gesture that matches any of the gesture names in `*completion-gestures*`.

- `:complete`—the function must complete the input as much as possible, except that if the user's input exactly matches one of the possibilities, even if it is a left substring of another possibility, the shorter possibility is returned as the result. This is the mode used when the user issues a delimiter or activation gesture that is not a partial completer.
- `:possibilities`—the function must return an alist of the possible completions as its fifth value. This is the mode used when the user a gesture that matches any of the gesture names in `*possibilities-gestures*` or `*help-gestures*` (if *help-displays-possibilities* is *true*).

function must return five values:

- *string*—the completed input string.
- *success*—*true* if completion was successful, otherwise *false*.
- *object*—the object corresponding to the completion, or `nil` if the completion was unsuccessful.
- *nmatches*—the number of possible completions of the input.
- *possibilities*—an alist of completions whose entries are a list of a string and an object, returned only when the completion mode is `:possibilities`. This list will be freshly created.

`complete-input` returns three values: *object*, *success*, and *string*. In addition, the printed representation of the completed input will be inserted into the input buffer of *stream* in place of the user-supplied string by calling `replace-input`.

partial-completers is a list of characters that delimit portions of a name that can be completed separately. The default is an empty list.

If the boolean *allow-any-input* is *true*, then `complete-input` will return as soon as the user issues an activation gesture, even if the input is not any of the possibilities. If the input is not one of the possibilities, the three values returned by `complete-input` will be `nil`, `t`, and the string. The default for *allow-any-input* is *false*.

If *possibility-printer* is supplied, it must be a function of three arguments, a possibility, a presentation type, and a stream; it has dynamic extent. The function displays the possibility on the stream. The possibility will be a list of two elements, the first being a string and the second being the object corresponding to the string.

If *help-display-possibilities* is *true* (the default), then when the user issues a help gesture (a gesture that matches one of the gesture names in `*help-gestures*`), CLIM will display all the matching possibilities. If it is *false*, then CLIM will not display the possibilities unless the user issues a possibility gesture (a gesture that matches one of the gesture names in `*possibilities-gestures*`).

⇒ `simple-completion-error` [*Condition*]

The error that is signalled by `complete-input` when no completion is found. This is a subclass of `simple-parse-error`.

⇒ `completing-from-suggestions` *(stream* `&key` *partial-completers allow-any-input possibility-printer (help-displays-possibilities* `t`*))* `&body` *body* [*Macro*]

Reads input from the *input editing stream stream*, completing over a set of possibilities generated by calls to `suggest` within *body*. *body* may have zero or more declarations as its first forms.

`completing-from-suggestions` returns three values, *object*, *success*, and *string*

The *stream* argument is not evaluated, and must be a symbol that is bound to a stream. If *stream* is `t` (the default), `*standard-input*` is used.

partial-completers, *allow-any-input*, and *possibility-printer* are as for `complete-input`.

Implementations will probably use `complete-from-generator` to implement this.

It is permitted for the function `suggest` to have lexical scope only within the body of `completing-from-suggestions`.

⇒ `suggest` *completion object* [*Function*]

Specifies one possibility for `completing-from-suggestions`. *completion* is a string, the printed representation of *object*. *object* is the internal representation.

It is permitted for this function to have lexical scope, and be defined only within the body of `completing-from-suggestions`.

⇒ `complete-from-generator` *string function delimiters* `&key` *(action* `:complete`*) predicate* [*Function*]

Given an input string *string* and a list of delimiter characters *delimiters* that act as partial completion characters, `complete-from-generator` completes against the possibilities that are generated by the function *generator*. *generator* is a function of two arguments, the string *string* and another function that it calls in order to process the possibility; it has dynamic extent.

action will be one of `:complete`, `:complete-maximal`, `:complete-limited`, or `:possibilities`. These are described under the function `complete-input`.

predicate must be a function of one argument, an object. If the predicate returns *true*, the possibility corresponding to the object is processed, otherwise it is not. It has dynamic extent.

`complete-from-generator` returns five values, the completed input string, the success value (*true* if the completion was successful, otherwise *false*), the object matching the completion (or `nil` if unsuccessful), the number of matches, and a list of possible completions if *action* was `:possibilities`.

This function is one that will typically be passed as the second argument to `complete-input`.

⇒ `complete-from-possibilities` *string completions delimiters* `&key` *(action* `:complete`*) predicate name-key value-key* [*Function*]

Given an input string *string* and a list of delimiter characters *delimiters* that act as partial completion characters, `complete-from-possibilities` completes against the possibilities in the sequence *completions*. The completion string is extracted from the possibilities in completions by applying *name-key*, which is a function of one argument. The object is extracted by applying *value-key*, which is a function of one argument. *name-key* defaults to `first`, and *value-key* defaults to `second`.

action will be one of `:complete`, `:complete-maximal`, `:complete-limited`, or `:possibilities`. These are described under the function `complete-input`.

predicate must be a function of one argument, an object. If the predicate returns *true*, the possibility corresponding to the object is processed, otherwise it is not.

predicate, *name-key*, and *value-key* have dynamic extent.

`complete-from-possibilities` returns five values, the completed input string, the success value (*true* if the completion was successful, otherwise *false*), the object matching the completion (or `nil` if unsuccessful), the number of matches, and a list of possible completions if *action* was `:possibilities`.

This function is one that will typically be passed as the second argument to `complete-input`.

⇒ `with-accept-help` *options* `&body` *body* [*Macro*]

Binds the dynamic environment to control the documentation produced by help and possibilities gestures during user input in calls to `accept` with the dynamic scope of *body*. *body* may have zero or more declarations as its first forms.

options is a list of option specifications. Each specification is itself a list of the form *(help-option help-string)*. *help-option* is either a symbol that is a *help-type* or a list of the form *(help-type mode-flag)*.

help-type must be one of:

- `:top-level-help`—specifies that *help-string* be used instead of the default help documentation provided by `accept`.
- `:subhelp`—specifies that *help-string* be used in addition to the default help documentation provided by `accept`.

mode-flag must be one of:

- `:append`—specifies that the current help string be appended to any previous help strings of the same help type. This is the default mode.

- `:override`—specifies that the current help string is the help for this help type; no lower-level calls to `with-accept-help` can override this. (`:override` works from the out-side in.)
- `:establish-unless-overridden`—specifies that the current help string be the help for this help type unless a higher-level call to `with-accept-help` has already established a help string for this help type in the `:override` mode. This is what `accept` uses to establish the default help.

help-string is a string or a function that returns a string. If it is a function, it receives three arguments, the stream, an action (either `:help` or `:possibilities`) and the help string generated so far.

None of the arguments is evaluated.

Chapter 25

Menu Facilities

Major issue: *There is a general issue about how these menus fit in with the menus that might be provided by the underlying toolkit. For example, under what circumstances is CLIM allowed to directly use the menu facilities provided by the host? Should* `:leave-menu-visible t` *interact with the "pushpin" facility provided by OpenLook? — SWM*

⇒ `menu-choose` *items* `&key` *associated-window printer presentation-type text-style foreground background default-item label cache unique-id id-test cache-value cache-test max-width max-height n-rows n-columns x-spacing y-spacing row-wise cell-align-x cell-align-y scroll-bars pointer-documentation* [*Generic Function*]

Displays a menu whose choices are given by the elements of the sequence *items*. It returns three values: the value of the chosen item, the item itself, and the pointer button event corresponding to the gesture that the user used to select it. If the user aborts out of the menu, a single value is returned, `nil`.

`menu-choose` will call `frame-manager-menu-choose` on the frame manager being used by *associated-window* (or the frame manager of the current application frame). All of the arguments to `menu-choose` will be passed on to `frame-manager-menu-choose`.

⇒ `frame-manager-menu-choose` *frame-manager items* `&key` *associated-window printer presentation-type text-style foreground background default-item label cache unique-id id-test cache-value cache-test max-width max-height n-rows n-columns x-spacing y-spacing row-wise cell-align-x cell-align-y scroll-bars pointer-documentation* [*Generic Function*]

Displays a menu whose choices are given by the elements of the sequence *items*. It returns three values: the value of the chosen item, the item itself, and the pointer button event corresponding to the gesture that the user used to select it. If the user aborts out of the menu, a single value is returned, `nil`.

Implementation note: the default method on `standard-frame-manager` will generally be implemented in terms of CLIM's own window stream and formatting facilities, such as using `menu-choose-from-drawer` on a stream allocated by `with-menu`. However, some frame managers may be able to use a native menu facility to handle most (if not all) menus. If the native menu facility cannot handle some cases, it can simply use `call-next-method` to invoke the default method.

items is a sequence of menu items. Each menu item has a visual representation derived from a display object, an internal representation that is a value object, and a set of menu item options. The form of a menu item is one of the following:

- An atom. The item is both the display object and the value object.
- A cons. The `car` is the display object and the `cdr` is the value object. The value object must be an atom. If you need to return a list as the value, use the :value option in the list menu item format described below.
- A list. The `car` is the display object and the cdr is a list of alternating option keywords and values. The value object is specified with the keyword `:value` and defaults to the display object if `:value` is not present.

The menu item options are:

- `:value`—specifies the value object.
- `:style`—specifies the text style used to `princ` the display object when neither *presentation-type* nor *printer* is supplied.
- `:items`—specifies a sequence of menu items for a sub-menu to be used if this item is selected.
- `:documentation`—associates some documentation with the menu item. When *:pointer-documentation* is not `nil`, this will be used as pointer documentation for the item.
- `:active`—when *true* (the default), this item is active. When *false*, the item is inactive, and cannot be selected. CLIM will generally provide some visual indication that an item is inactive, such as by "graying over" the item.
- `:type`—specifies the type of the item. `:item` (the default) indicates that the item is a normal menu item. `:label` indicates that the item is simply an inactive label; labels will not be "grayed over". `:divider` indicates that the item serves as a divider between groups of other items; divider items will usually be drawn as a horizontal line.

The visual representation of an item depends on the *printer* and *presentation-type* keyword arguments. If *presentation-type* is supplied, the visual representation is produced by `present` of the menu item with that presentation type. Otherwise, if *printer* is supplied, the visual representation is produced by the *printer* function, which receives two arguments, the *item* and a *stream* to do output on. The *printer* function should output some text or graphics at the stream's cursor position, but need not call `present`. If neither *presentation-type* nor *printer* is supplied, the visual representation is produced by `princ` of the display object. Note that if *presentation-type* or *printer* is supplied, the visual representation is produced from the entire menu item, not just from the display object. CLIM implementations are free to use the menus provided by the underlying window system when possible; this is likely to be the case when the printer and presentation-type are the default, and no other options are supplied.

associated-window is the CLIM window with which the menu is associated. This defaults to the top-level window of the current application frame.

default-item is the menu item where the mouse will appear.

text-style is a text style that defines how the menu items are presented. *foreground* and *background* specify the foreground and background ink of the menu; they default to the foreground and background ink of *associated-window*.

label is a string to which the menu title will be set.

printer is a function of two arguments used to print the menu items in the menu. The two arguments are the menu item and the stream to output it on. It has dynamic extent.

presentation-type specifies the presentation type of the menu items.

cache is a boolean that indicates whether CLIM should cache this menu for later use. (Caching menus might speed up later uses of the same menu.) If *cache* is *true*, then *unique-id* and *id-test* serve to uniquely identify this menu. When cache is *true*, *unique-id* defaults to *items*, but programmers will generally wish to specify a more efficient tag. *id-test* is a function of two arguments used to compare unique-ids, which defaults to `equal`. *cache-value* is the value that is used to indicate that a cached menu is still valid. It defaults to *items*, but programmers may wish to supply a more efficient cache value than that. *cache-test* is a function of two arguments that is used to compare cache values, which defaults to `equal`. Both *cache-value* and *unique-id* have dynamic extent.

max-width and *max-height* specify the maximum width and height of the menu, in device units. They can be overridden by *n-rows* and *n-columns*.

n-rows and *n-columns* specify the number of rows and columns in the menu.

x-spacing specifies the amount of space to be inserted between columns of the table; the default is the width of a space character. It is specified the same way as the `:x-spacing` option to `formatting-table`.

y-spacing specifies the amount of blank space inserted between rows of the table; the default is the vertical spacing for the stream. The possible values for this option are the same as for the *:y-spacing* option to `formatting-table`.

cell-align-x specifies the horizontal placement of the contents of the cell. Can be one of `:left`, `:right`, or `:center`. The default is `:left`. The semantics are the same as for the `:align-x` option to `formatting-cell`.

cell-align-y specifies the vertical placement of the contents of the cell. Can be one of `:top`, `:bottom`, or `:center`. The default is `:top`. The semantics are the same as for the `:align-y` option to `formatting-cell`.

row-wise is as for `formatting-item-list`. It defaults to `t`.

scroll-bars specifies whether the menu should have scroll bars. It acts the same way as the `:scroll-bars` option to `make-clim-stream-pane`. It defaults to `:vertical`.

pointer-documentation is either `nil` (the default), meaning that no pointer documentation should be computed, or a stream on which pointer documentation should be displayed.

⇒ `menu-choose-from-drawer` *menu presentation-type drawer* `&key` *x-position y-position cache unique-id id-test cache-value cache-test default-presentation pointer-documentation* [*Generic Function*]

This is a a lower-level routine for displaying menus. It allows the programmer much more flexibility in the menu layout. Unlike `menu-choose`, which automatically creates and lays out the menu, `menu-choose-from-drawer` takes a programmer-provided window and drawing function. The drawing function is responsible for drawing the contents of the menu; generally it will be a lexical closure that closes over the menu items.

`menu-choose-from-drawer` draws the menu items into that window using the drawing function. The drawing function gets called with two arguments, *stream* and *presentation-type*. It can use *presentation-type* for its own purposes, such as using it as the presentation type argument in a call to `present`.

`menu-choose-from-drawer` returns two values: the object the user clicked on, and the pointer button event. If the user aborts out of the menu, a single value is returned, `nil`.

menu is a CLIM window to use for the menu. This argument may be specialized to provide a different look-and-feel for different host window systems.

presentation-type is a presentation type specifier for each of the mouse-sensitive items in the menu. This is the input context that will be established once the menu is displayed. For programmers who don't need to define their own types, a useful presentation type is `menu-item`.

drawer is a function that takes two arguments, *stream* and *presentation-type*, draws the contents of the menu. It has dynamic extent.

x-position and *y-position* are the requested x and y positions of the menu. They may be `nil`, meaning that the position is unspecified.

If *leave-menu-visible* is *true*, the window will not be deexposed once the selection has been made. The default is *false*, meaning that the window will be deexposed once the selection has been made.

default-presentation is used to identify the presentation that the mouse is pointing to when the menu comes up.

cache, *unique-id*, *id-test*, *cache-value*, and *cache-test* are as for `menu-choose`.

⇒ `draw-standard-menu` *stream presentation-type items default-item* `&key` *item-printer max-width max-height n-rows n-columns x-spacing y-spacing row-wise cell-align-x cell-align-y* [*Function*] `draw-standard-menu` is the function used by CLIM to draw the contents of a menu, unless the current frame manager determines that host window toolkit should be used to draw the menu instead. *stream* is the stream onto which to draw the menu, *presentation-type* is the presentation type to use for the menu items (usually `menu-item`), and *item-printer* is a function used to draw each item. *item-printer* defaults to `print-menu-item`.

items, *default-item*, *max-width*, *max-height*, *n-rows*, *n-columns*, *x-spacing*, *y-spacing*, *row-wise*, *cell-align-x*, and *cell-align-y* are as for `menu-choose`

⇒ `print-menu-item` *menu-item* `&optional` *(stream* `*standard-output*`*)* [*Function*]

Given a menu item *menu-item*, displays it on the stream *stream*. This is the function that `menu-choose` uses to display menu items if no printer is supplied.

⇒ `menu-item-value` *menu-item* [*Function*]

Returns the value of the menu item *menu-item*, where the format of a menu item is described under `menu-choose`. If *menu-item* is not a menu item, the result is unspecified.

⇒ `menu-item-display` *menu-item* [*Function*]

Returns the display object of the menu item *menu-item*, where the format of a menu item is described under `menu-choose`. If *menu-item* is not a menu item, the result is unspecified.

⇒ `menu-item-options` *menu-item* [*Function*]

Returns the options of the menu item *menu-item*, where the format of a menu item is described under `menu-choose`. If *menu-item* is not a menu item, the result is unspecified.

⇒ `with-menu` *(menu* `&optional` *associated-window* `&key` *(deexpose t))* `&body` *body* [*Macro*]

Binds *menu* to a "temporary" window, exposes the window on the same screen as the *associated-window* and runs the body. After the body has been run, the window is deexposed only if the boolean *deexpose* is *true* (the default).

The values returned by `with-menu` are the values returned by *body*. *body* may have zero or more declarations as its first forms.

menu must be a variable name. *associated-window* is as for `menu-choose`.

None of the arguments is evaluated.

Chapter 26

Dialog Facilities

Major issue: *There is a general issue about how these dialogs fit in with the dialogs that might be provided by the underlying toolkit. For example, under what circumstances is CLIM allowed to directly use the dialog facility provided by the host? — SWM*

⇒ `accepting-values` *(*`&optional` *stream* `&key` *own-window exit-boxes text-style foreground background initially-select-query-identifier modify-initial-query resynchronize-every-pass resize-frame align-prompts label scroll-bars x-position y-position width height command-table frame-class)* `&body` *body* [*Macro*]

Builds a dialog for user interaction based on calls to `accept` within *body*. The user can select the values and change them, or use defaults if they are supplied. The dialog will also contain some sort of "end" and "abort" choices. If "end" is selected, then `accepting-values` returns whatever values the body returns. If "abort" is selected, `accepting-values` will invoke the `abort` restart.

stream is an interactive stream that `accepting-values` will use to build up the dialog. The *stream* argument is not evaluated, and must be a symbol that is bound to a stream. If *stream* is `t` (the default), `*standard-input*` is used.

body is the body of the dialog, which contains calls to `accept` that will be intercepted by `accepting-values` and used to build up the dialog. *body* may have zero or more declarations as its first forms.

An `accepting-values` dialog is implemented as an application frame with a looping structure. First, *body* is evaluated in order to collect the output. While the body is being evaluated, all calls to `accept` call the `accept-present-default` presentation methods instead of calling the `accept` presentation methods. The output is then displayed, preferably using incremental redisplay in order to avoid unnecessary redisplay of unchanged output. If *align-prompts* is *true* (the default is `nil`), then the fields of the dialog will be displayed within a call to `formatting-table` so that the prompts are aligned vertically on their right-hand sides and the input fields are aligned on their left-hand sides. This option is intended to support toolkits where users expect dialogs to have this sort of layout.

After `accepting-values` has displayed all of the fields, it awaits a user gesture, such as clicking on one of the fields of the dialog. When the user clicks on a field, `accepting-values` reads a new value for that field using `accept` and replaces the old value with the new value. Then the loop is started again, until the user either exits or aborts from the dialog.

Because of its looping structure, `accepting-values` needs to be able to uniquely identify each call to `accept` in the body of the dialog. The *query identifier* is used to identify the calls to `accept`. The query identifier for a call to `accept` is computed on each loop through the dialog, and should therefore be free of side-effects. Query identifiers are compared using `equal`. Inside of `accepting-values`, programmers should supply the `:query-identifier` argument to each call to `accept`. If `:query-identifier` is not explicitly supplied, the prompt for that call to `accept` is used as the query identifier. Thus, if `:query-identifier` is not supplied, programmers must ensure that all of the prompts are different. If there is more than one call to `accept` with the same query identifier, the behavior of `accepting-values` is unspecified.

While inside `accepting-values`, calls to `accept` return a third value, a boolean ("changed-p") that indicates whether the object is the result of new input by the user, or is just the previously supplied default. The third value will be *true* in the former case, *false* in the latter.

Implementation note: each invocation of `accepting-values` will probably need to maintain a table that maps from a query identifier to the output record for the field that used the query identifier, and the output record for each field in the dialog will probably need a mapping back to the query identifier. A mediating object (a "query object") is also useful, for instance, as a place to store the "changed-p" flag.

The class of the application frame created by `accepting-values` will be `accept-values` or a subclass of `accept-values`. Programmers can use a class of their own by supplying the name of a class via the *frame-class* argument. CLIM will use the command table `accept-values` as the command table for `accepting-values`. Programmers can supply a command table of their own by supplying the *command-table* argument.

When *own-window* is non-`nil`, the dialog will appear in its own "popped-up" window. In this case the initial value of *stream* is a window with which the dialog is associated. (This is similar to the *associated-window* argument to `menu-choose`.) Within the *body*, the value of *stream* will be the "popped-up" window. *own-window* is either `t` or a list of alternating keyword options and values. The accepted options are `:right-margin` and `:bottom-margin`; their values control the amount of extra space to the right of and below the dialog (useful if the user's responses to the dialog take up more space than the initially displayed defaults). The allowed values for `:right-margin` are the same as for the `:x-spacing` option to `formatting-table`; the allowed values for `:bottom-margin` are the same as for the `:y-spacing` option.

Minor issue: *When the programmer supplies* `:right-margin` *or* `:bottom-margin` *options in the own-window argument, how is he supposed to determine what's needed? How about providing an option to permit the window to resize itself dynamically? There really needs to be a hook into* `note-space-requirements-changed` *or something. — barmar, SWM*

text-style is the default text style for the dialog. *foreground* and *background* specify the foreground and background ink of an "own window" dialog; they default to the foreground and background ink of *stream*.

exit-boxes specifies what the exit boxes should look like. The default behavior is though the following were supplied:

```
'((:exit "<End> uses these values")
  (:abort "<Abort> aborts"))
```

Minor issue: *We need to describe the interpretation of the exit-boxes argument. Are other keywords beside* `:exit` *and* `:abort` *permitted, such as* `:help`*? It's pretty common for a dialog to have multiple ways to exit; perhaps* `accepting-values` *should return a second value that indicates which exit box was selected. This alist looks sort of like a menu item list; perhaps the full generality should be permitted (so that the style of the exit box messages can be specified). The text strings that are shown in the default value look more like documentation than button labels; I think both are necessary, and the programmer must be able to find out what the default labels are so that he can include them in the documentation (rather than hard-coding "¡End¿" and "¡Abort¿"). — barmar*

initially-select-query-identifier specifies that a particular field in the dialog should be pre-selected when the user interaction begins. The field to be selected is tagged by the `:query-identifier` option to `accept`. When the initial display is output, the input editor cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it. The default value, if any, for the selected field is not displayed. When *modify-initial-query* is *true*, the initially selected field is selected for modification rather than for replacement; the default is `nil`.

resynchronize-every-pass is a boolean option specifying whether earlier queries depend on later values; the default is *false*. When it is *true*, the contents of the dialog are redisplayed an additional time after each user interaction. This has the effect of ensuring that, when the value of some field of a dialog depends on the value of another field, all of the displayed fields will be up to date.

When *resize-frame* is *true*, own-window dialogs will be resized after each pass through the redisplay loop. The default is `nil`.

label is as for `menu-choose`. *x-position* and *y-position* are as for `menu-choose-from-drawer`. *width* and *height* are real numbers that specify the initial width and height of own-window dialogs.

⇒ `accept-values` [*Application Frame*]

`accepting-values` must be implemented as a CLIM application frame that uses `accept-values` as the name of the frame class.

⇒ `display-exit-boxes` *frame stream view* [*Generic Function*]

Displays the exits boxes for the `accepting-values` frame *frame* on the stream *strea,* in the view *view.* The exit boxes specification is not passed in directly, but is a slot in the frame. The default method (on `accept-values`) simply writes a line of text associating the Exit and Abort strings with presentations that either exit or abort from the dialog.

The *frame*, *stream*, and *view* arguments may be specialized to provide a different look-and-feel for different host window systems.

⇒ `accept-values-resynchronize` *stream* [*Generic Function*]

Causes `accepting-values` to resynchronizes the dialog once on the accepting values stream *stream* before it restarts the dialog loop.

⇒ `accept-values-command-button` *(*`&optional` *stream* `&key` *documentation query-identifier cache-value cache-test resynchronize) prompt* `&body` *body* [*Macro*]

Displays the prompt *prompt* on the stream *stream* and creates an area (the "button"). When a pointer button is clicked in this area at runtime, *body* will be evaluated.

`accept-values-command-button` must be implemented by expanding into a call to `invoke-accept-values-command-button`, supplying a function that executes *body* as the *continuation* argument to `accept-values-command-button`.

The *stream* argument is not evaluated, and must be a symbol that is bound to a stream. If *stream* is `t` (the default), `*standard-input*` is used. *body* may have zero or more declarations as its first forms.

⇒ `invoke-accept-values-command-button` *stream continuation view prompt* `&key` *documentation query-identifier cache-value cache-test resynchronize* [*Method*]

Displays the prompt *prompt* on the stream *stream* and creates an area (the "button"). When a pointer button is clicked in this area at runtime, the continuation will be called. *continuation* is a function that takes no arguments. *view* is a view.

prompt may be either a string (which will be displayed via `write-string`), or a form that will be evaluated to draw the button.

documentation is an object that will be used to produce pointer documentation for the button. It defaults to *prompt*. If it is a string, the string itself will be used as the pointer documentation. Otherwise it must be a function of one argument, the stream to which the documentation should be written.

When *resynchronize* is *true*, the dialog will be redisplayed an additional time whenever the command button is clicked on. See the *resynchronize-every-pass* argument to `accepting-values`.

cache-value and *cache-test* are as for `updating-output`. That is, *cache-value* should evaluate to the same value if and only if the output produced by *prompt* does not ever change. *cache-test* is a function of two arguments that is used to compare cache values. *cache-value* defaults to `t` and *cache-test* defaults to `eql`.

This function may only be used inside the dynamic context of an `accepting-values`.

Part VII

Building Applications

Chapter 27

Command Processing

27.1 Commands

A *command* is an object that represents a user interaction. Commands are stored as a cons of the command name and a list of the command's arguments. All positional arguments will be represented in the command object, but only those keywords arguments that were explicitly supplied by the user will be included. When the first element of the cons is `apply`'ed to the rest of the cons, the code representing that interaction is executed.

A *partial command* is a command object with the value of `*unsupplied-argument-marker*` in place of any argument that needs to be filled in.

Every command is named by *command name*, which is a symbol. To avoid collisions among command names, application frames should reside in their own package; for example, the `com-show-chart` command might be defined for both a spreadsheet and a medical application.

⇒ `command-name` *command* [*Function*]

Given a command object *command*, returns the command name.

⇒ `command-arguments` *command* [*Function*]

Given a command object *command*, returns the command's arguments.

⇒ `partial-command-p` *command* [*Function*]

Returns *true* if the *command* is a partial command, that is, has any occurrences of `*unsupplied-argument-marker*` in it. Otherwise, `partial-command-p` returns *false*.

⇒ `define-command` *name-and-options arguments* `&body` *body* [*Macro*]

This is the most basic command-defining form. Usually, the programmer will not use `define-`

`command` directly, but will instead use a `define-`*frame*`-command` form that is automatically generated by `define-application-frame`. `define-`*frame*`-command` adds the command to the application frame's command table. By default, `define-command` does not add the command to any command table.

name-and-options is either a command name, or a cons of the command name and a list of keyword-value pairs.

`define-command` defines two functions. The first function has the same name as the command name, and implements the body of the command. It takes as arguments the arguments to the command as specified by the `define-command` form, as required and keyword arguments.

The name of the other function defined by `define-command` is unspecified. It implements the code used by the command processor for parsing and returning the command's arguments.

The keywords from *name-and-options* can be:

- `:command-table` *command-table-name*, where *command-table-name* either names a command table to which the command will be added, or is `nil` (the default) to indicate that the command should not be added to any command table. If the command table does not exist, the `command-table-not-found` error will be signalled. This keyword is only accepted by `define-command`, not by `define-`*frame*`-command`.
- `:name` *string*, where *string* is a string that will be used as the command-line name for the command for keyboard interactions in the command table specified by the `:command-table` option. The default is `nil`, meaning that the command will not be available via command-line interactions. If *string* is `t`, then the command-line name will be generated automatically, as described in `add-command-to-command-table`.
- `:menu` *menu-spec*, where *menu-spec* describes an item in the menu of the command table specified by the `:command-table` option. The default is `nil`, meaning that the command will not be available via menu interactions. If *menu-spec* is a string, then that string will be used as the menu name. If *menu-spec* is `t`, then if a command-line name was supplied, it will be used as the menu name; otherwise the menu name will be generated automatically, as described in `add-command-to-command-table`. Otherwise, *menu-spec* must be a cons of the form `(`*string* `.` *menu-options*`)`, where *string* is the menu name and *menu-options* consists of keyword-value pairs. The valid keywords are `:after`, `:documentation`, and `:text-style`, which are interpreted as for `add-menu-item-to-command-table`.
- `:keystroke` *gesture*, where *gesture* is a keyboard gesture name that specifies a keystroke accelerator to use for this command in the command table specified by the `:command-table` option. The default is `nil`, meaning that there is no keystroke accelerator.

The `:name`, `:menu`, and `:keystroke` options are only allowed if the `:command-table` option was supplied explicitly or implicitly, as in `define-`*frame*`-command`.

arguments is a list consisting of argument descriptions. A single occurrence of the symbol `&key` may appear in *arguments* to separate required command arguments from keyword arguments. Each argument description consists of a parameter variable, followed by a presentation type specifier, followed by keyword-value pairs. The keywords can be:

- `:default` *value*, where *value* is the default that should be used for the argument, as for `accept`.
- `:default-type` is the same as for `accept`.
- `:display-default` is the same as for `accept`.
- `:mentioned-default` *value*, where *value* is the default that should be used for the argument when a keyword is explicitly supplied via the command-line processor, but no value is supplied for it. `:mentioned-default` is only allowed on keyword arguments.
- `:prompt` *string*, where *string* is a prompt to print out during command-line parsing, as for `accept`.
- `:documentation` *string*, where *string* is a documentation string that describes what the argument is.
- `:when` *form*. *form* is evaluated in a scope where the parameter variables for the required parameters are bound, and if the result is `nil`, the keyword argument is not available. `:when` is only allowed on keyword arguments, and *form* cannot use the values of other keyword arguments.
- `:gesture` *gesture*, where *gesture* is either a pointer gesture name or a list of a pointer gesture name followed by keyword-value pairs. When a gesture is supplied, a presentation translator will be defined that translates from this argument's presentation type to an instance of this command with the selected object as the argument; the other arguments will be filled in with their default values. The keyword-value pairs are used as options for the translator. Valid keywords are `:tester`, `:menu`, `:priority`, `:echo`, `:documentation`, and `:pointer-documentation`. The default for *gesture* is `nil`, meaning no translator will be written. `:gesture` is only allowed when the `:command-table` option was supplied to the command-defining form.

body implements the body of the command. It has lexical access to all of the commands arguments. If the body of the command needs access to the application frame itself, it should use `*application-frame*`. The returned values of body are ignored. *body* may have zero or more declarations as its first forms.

`define-command` must arrange for the function that implements the body of the command to get the proper values for unsupplied keyword arguments.

name-and-options and *body* are not evaluated. In the argument descriptions, the parameter variable name is not evaluated, and everything else is evaluated at run-time when argument parsing reaches that argument, except that the value for `:when` is evaluated when parsing reaches the keyword arguments, and `:gesture` isn't evaluated at all.

27.2 Command Tables

There are four main styles of interaction: keyboard interaction using a command- line processor, keyboard interaction using keystroke accelerators, mouse interaction via command menus, and mouse interaction via translators. A *command table* is an object that serves to mediate between an application frame, a set of commands, and the four interaction styles. Command tables contain the following information:

- The name of the command table, which is a symbol.
- An ordered list of command tables to inherit from.
- The set of commands that are present in this command table.
- A table that associates command-line names to command names (used to support command-line processor interactions).
- A set of presentation translators, defined via `define-presentation-translator` and `define-presentation-to-command-translator`.
- A table that associates keyboard gesture names to menu items (used to support keystroke accelerator interactions). The keystroke accelerator table does not contain any items inherited from superior command tables.
- A menu that associates menu names to command menu items (used to support interaction via command menus). The command menu items can invoke commands or sub-menus. By default, the menu does not contain any command menu items inherited from superior command tables, although this can be overridden by the `:inherit-menu` option to `define-command-table`.

We say that a command is *present* in a command table when it has been added to that command table. We say that a command is *accessible* in a command table when it is present in that command table or is present in any of the command tables from which that command table inherits.

⇒ `command-table` [*Protocol Class*]

The protocol class that corresponds to command tables. If you want to create a new class that behaves like a command table, it should be a subclass of `command-table`. All instantiable subclasses of `command-table` must obey the command table protocol. Members of this class are mutable.

⇒ `command-table-p` *object* [*Protocol Predicate*]

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

⇒ `standard-command-table` [*Class*]

The instantiable class that implements command tables, a subclass of `command-table`. `make-command-table` returns objects that are members of this class.

Minor issue: *Do we really want to advertise these classes, since all the functions below are vanilla functions instead of generic functions? Or should we make those functions be generic functions? — SWM*

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

Returns the name of the *command table command-table*.

⇒ `command-table-inherit-from` *command-table* [*Generic Function*]

Returns a list of the command tables from which the *command table command-table* inherits. This function returns objects that reveal CLIM's internal state; do not modify those objects.

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

Defines a command table whose name is the symbol *name*. The new command table inherits from all of the command tables specified by *inherit-from*, which is a list of *command table designators* (that is, either a command table or a symbol that names a command table). The inheritance is done by union with shadowing. If no inheritance is specified, the command table will be made to inherit from CLIM's global command table. (This command table contains such things as the "menu" translator that is associated with the right-hand button on pointers.)

If *inherit-menu* is *true*, the new command table will inherit the menu items and keystroke accelerators from all of the inherited command tables. If it is *false* (the default), no menu items or keystroke accelerators will be inherited.

menu can be used to specify a menu for the command table. The value of *menu* is a list of clauses. Each clause is a list with the syntax (*string type value* `&key` *keystroke documentation text-style*), where *string*, *type*, *value*, *keystroke*, *documentation*, and *text-style* are as for `add-menu-item-to-command-table`.

If the command table named by *name* already exists, `define-command-table` will modify the existing command table to have the new value for *inherit-from* and *menu*, and leaves the other attributes for the existing command table alone.

None of `define-command-table`'s arguments are evaluated.

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

Creates a command table named *name*. *inherit-from*, *menu*, and *inherit-menu* are the same as for `define-command-table`. `make-command-table` does not implicitly include CLIM's global command table in the inheritance list for the new command table. If the command table already exists and *errorp* is *true*, the `command-table-already-exists` error will be signalled. If the command table already exists and *errorp* is *false*, then the old command table will be discarded. The returned value is the command table.

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

Returns the command table named by *name*. If *name* is itself a command table, it is returned. If the command table is not found and *errorp* is *true*, the `command-table-not-found` error will be signalled.

⇒ `command-table-error` [*Error Condition*]

The class that is the superclass of the following four conditions. This class is a subclass of `error`.

`command-table-error` and its subclasses must handle the `:format-string` and `:format-arguments` initargs, which are used to specify a control string and arguments for a call to `format`.

⇒ `command-table-not-found` *[Error Condition]*

The error that is signalled by such functions as `find-command-table` when a command table is not found.

⇒ `command-table-already-exists` *[Error Condition]*

The error that is signalled when the programmer tries to create a command table that already exists.

⇒ `command-not-present` *[Error Condition]*

The error that is signalled when a command is not present in a command table.

⇒ `command-not-accessible` *[Error Condition]*

The error that is signalled when a command is not accessible in a command table.

⇒ `command-already-present` *[Error Condition]*

The error that is signalled when a function tries to add a command to a command table when it is already present in the command table.

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

Adds the command named by *command-name* to the command table specified by the *command table designator command-table*.

name is the command-line name for the command, and can be `nil`, `t`, or a string. When it is `nil`, the command will not be available via command-line interactions. When it is a string, that string is the command-line name for the command. When it is `t`, the command-line name is generated automatically by calling `command-name-from-symbol` on *command-name*. For the purposes of command-line name lookup, the character case of *name* is ignored.

menu is a menu item for the command, and can be `nil`, `t`, a string, or a cons. When it is `nil`, the command will not be available via menus. When it is a string, the string will be used as the menu name. When *menu* is `t` and *name* is a string, then *name* will be used as the menu name. When *menu* is `t` and *name* is not a string, an automatically generated menu name will be used. When *menu* is a cons of the form `(`*string* `.` *menu-options*`)`, *string* is the menu name and *menu-options* consists of keyword-value pairs. The valid keywords are `:after`, `:documentation`, and `:text-style`, which are interpreted as for `add-menu-item-to-command-table`.

The value for *keystroke* is either keyboard gesture name or `nil`. When it is a gesture name, it is the keystroke accelerator for the command; otherwise the command will not be available via keystroke accelerators.

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

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

Removes the command named by *command-name* from the command table specified by the *command table designator command-table.*

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

⇒ `command-name-from-symbol` *symbol* [*Function*]

Generates a string suitable for use as a command-line name from the symbol *symbol.* The string consists the symbol name with the hyphens replaced by spaces, and the words capitalized. If the symbol name is prefixed by "COM-", the prefix is removed. For example, if the symbol is `com-show-file`, the result string will be "Show File".

⇒ `do-command-table-inheritance` *(command-table-var command-table)* `&body` *body* [*Macro*]

Successively executes *body* with *command-table-var* bound first to the command table specified by the *command table designator command-table*, and then (recursively) to all of the command tables from which *command-table* inherits.

The *command-table-var* argument is not evaluated. *body* may have zero or more declarations as its first forms.

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

Applies *function* to all of the commands accessible in the command table specified by the *command table designator command-table. function* must be a function that takes a single argument, the command name; it has dynamic extent.

If *inherited* is *false*, this applies *function* only to those commands 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-commands` returns `nil`.

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

Applies *function* to all of the command-line name accessible in the command table specified by the *command table designator command-table. function* must be a function of two arguments, the command-line name and the command name; it has dynamic extent.

If *inherited* is *false*, this applies *function* only to those command-line names 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-names` returns `nil`.

⇒ `command-present-in-command-table-p` *command-name command-table* [*Function*]

Returns *true* if the command named by *command-name* is present in the command table specified by the *command table designator command-table*, otherwise returns *false*.

⇒ `command-accessible-in-command-table-p` *command-name command-table* [*Function*]

If the command named by *command-name* is not accessible in the command table specified by the *command table designator command-table*, then this function returns `nil`. Otherwise, it returns the command table in which the command was found.

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

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

`find-command-from-command-line-name` ignores character case.

⇒ `command-line-name-for-command` *command-name command-table* `&key` *(errorp t)* [*Function*]

Returns the command-line name for *command-name* as it is installed in *command-table*. *command-table* is a *command table designator*.

If the command is not accessible in *command-table* or has no command-line name, then there are three possible results. If *errorp* is `nil`, then the returned value will be `nil`. If *errorp* is `:create`, then a command-line name will be generated, as described in `add-command-to-command-table`. Otherwise, if *errorp* is `t`, then the `command-not-accessible` error will be signalled. The returned command-line name should not be modified.

This is the inverse of `find-command-from-command-line-name`. It should be implemented in such as way that it is fast, since it may be used by presentation translators to produce pointer documentation.

⇒ `command-table-complete-input` *command-table string action* `&key` *frame* [*Function*]

A function that can be used as in conjunction with `complete-input` in order to complete over all of the command lines names accessible in the *command table command-table*. *string* is the input string to complete over, and *action* is as for `complete-from-possibilities`.

frame is either an application frame, or `nil`. If *frame* is supplied, no disabled commands should be offered as valid completions.

`command-table-complete-input` could be implemented by collecting all of the command line names accessible in the command table and then calling `complete-from-possibilities`, or it could be implemented more efficiently than that (such as by caching a sorted list of command line names and using a binary search).

⇒ `global-command-table` [*Command Table*]

The command table from which all other command tables inherit by default. Programmers should not explicitly add anything to or remove anything from this command table. CLIM can use this command to store internals or system-wide commands and translators (for example, the translator that implements the "identity" translation from a type to itself). Programmers should not casually install any commands or translators into this command table.

⇒ `user-command-table` [*Command Table*]

A command table that can be used by the programmer for any purpose. CLIM does not use it for anything, and its contents are completely undefined.

27.3 Command Menus

Each command table may have a menu consisting of an ordered sequence of command menu items. The menu specifies a mapping from a menu name (the name displayed in the menu) to a command menu item. The menu of an application frame's top-level command table may be presented in a window system specific way, for example, as a menu bar.

Command menu items are stored as a list of the form (*type value* . *options*), where *type* and *value* are as in `add-menu-item-to-command-table`, and *options* is a list of keyword-value pairs. The allowable keywords are `:documentation`, which is used to supply optional pointer documention for the command menu item, and `:text-style`, which is used to indicate what text style should be used for this command menu item when it is displayed in a command menu.

`add-menu-item-to-command-table`, `remove-menu-item-from-command-table`, and `find-menu-item` ignore the character case of the command menu item's name when searching through the command table's menu.

⇒ `add-menu-item-to-command-table` *command-table string type value* `&key` *documentation (after ':end) keystroke text-style (errorp t)* [*Function*]

Adds a command menu item to *command-table*'s menu. *string* is the name of the command menu item; its character case is ignored. *type* is either `:command`, `:function`, `:menu`, or `:divider`. *command-table* is a *command table designator*.

Minor issue: *How do we make iconic command menus? Probably another keyword... —SWM*

When *type* is `:command`, *value* must be a command (a cons of a command name followed by a list of the command's arguments), or a command name. (When *value* is a command name, it behaves as though a command with no arguments was supplied.) In the case where all of the command's required arguments are supplied, clicking on an item in the menu invokes the command immediately. Otherwise, the user will be prompted for the remaining required arguments.

When *type* is `:function`, *value* must be function having indefinite extent that, when called, returns a command. The function is called with two arguments, the gesture the user used to select the item (either a keyboard or button press event) and a "numeric argument".

When *type* is `:menu`, this item indicates that a sub-menu will be invoked, and so *value* must be another command table or the name of another command table.

When *type* is `:divider`, some sort of a dividing line is displayed in the menu at that point. If *string* is supplied, it will be drawn as the divider instead of a line. If the look and feel provided by the underlying window system has no corresponding concept, `:divider` items may be ignored. *value* is ignored.

documentation is a documentation string, which can be used as mouse documentation for the command menu item.

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

after must be either `:start` (meaning to add the new item to the beginning of the menu), `:end` or `nil` (meaning to add the new item to the end of the menu), or a string naming an existing entry (meaning to add the new item after that entry). If *after* is `:sort`, then the item is inserted in such as way as to maintain the menu in alphabetical order.

If *keystroke* is supplied, the item will be added to the command table's keystroke accelerator table. The value of *keystroke* must be a keyboard gesture name. This is exactly equivalent to calling `add-keystroke-to-command-table` with the arguments *command-table*, *keystroke*, *type* and *value*. When *keystroke* is supplied and *type* is `:command` or `:function`, typing a key on the keyboard that matches to the keystroke accelerator gesture will invoke the command specified by *value*. When *type* is `:menu`, the command will continue to be read from the sub-menu indicated by *value* in a window system specific manner.

If the item named by *string* is already present in the command table's menu and *errorp* is *true*, then the `command-already-present` error will be signalled. When the item is already present in the command table's menu and *errorp* is *false*, the old item will first be removed from the menu. Note that the character case of *string* is ignored when searching the command table's menu.

⇒ `remove-menu-item-from-command-table` *command-table string* `&key` *(errorp t)* [*Function*]

Removes the item named by *string* from *command-table*'s menu. *command-table* is a *command table designator*.

If the item is not present in the command table's menu and *errorp* is *true*, then the `command-not-present` error will be signalled. Note that the character case of *string* is ignored when searching the command table's menu.

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

Applies *function* to all of the items in *command-table*'s menu. *function* must be a function of three arguments, the menu name, the keystroke accelerator gesture (which will be `nil` if there is none), and the command menu item; it has dynamic extent. The command menu items are mapped over in the order specified by `add-menu-item-to-command-table`. *command-table* is a *command table designator*.

`map-over-command-table-menu-items` 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-menu-items` returns `nil`.

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

Given a menu name and a command table, returns two values, the command menu item and the command table in which it was found. (Since menus are not inherited, the second returned value will always be *command-table*.) *command-table* is a *command table designator*. This function returns objects that reveal CLIM's internal state; do not modify those objects.

If there is no command menu item corresponding to *menu-name* present in *command-table* and *errorp* is *true*, then the `command-not-accessible` error will be signalled. Note that the character case of *string* is ignored when searching the command table's menu.

⇒ `command-menu-item-type` *menu-item* [*Function*]

Returns the type of the command menu item *menu-item*, for example, `:menu` or `:command`. If *menu-item* is not a command menu item, the result is unspecified.

⇒ `command-menu-item-value` *menu-item* [*Function*]

Returns the value of the command menu item *menu-item*. For example, if the type of *menu-item* is `:command`, this will return a command or a command name. If *menu-item* is not a command menu item, the result is unspecified.

⇒ `command-menu-item-options` *menu-item* [*Function*]

Returns a list of the options for the command menu item *menu-item*. If *menu-item* is not a command menu item, the result is unspecified.

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

Displays *command-table*'s menu on *stream*. Implementations may choose to use `formatting-item-list` or may display the command table's menu in a platform dependent manner, such as using the menu bar on a Macintosh. *command-table* is a *command table designator*.

max-width, *max-height*, *n-rows*, *n-columns*, *x-spacing*, *y-spacing*, *row-wise*, *initial-spacing*, *cell-*