Common Lisp Interface Manager
Release 2.0 Specification

Scott McKay (SWM@Riverside.SCRC.Symbolics.COM)
William York (York@Fuji.ILA.COM)
with contributions by
John Aspinall (JGA@Riverside.SCRC.Symbolics.COM)
Dennis Doughty (Doughty@Fuji.ILA.COM)
Charles Hornig (Hornig@Riverside.SCRC.Symbolics.COM)
Richard Lamson (RSL@Fuji.ILA.COM)
David Linden (DCPL@Fuji.ILA.COM)
David Moon (Moon@Cambridge.Apple.COM)
Ramana Rao (rao@PARC.Xerox.COM)

April 2, 1992

Contents

Acknowledgments

The process of designing and implementing CLIM and writing the CLIM specification has been a long and sometimes bumpy one. The first thanks have to go to the people and organizations who allowed the people working on CLIM to continue to do so, despite the obstacles. In particular, Mark Son-Bell of ILA; Bob Laddaga and Saiid Zarrabian at Symbolics (and before or with them, Ken Sinclair, Rick Karash, and Mark Graffam); and Jim Vietch and Hanoch Eiron of Franz, Inc. have struggled with various and sometimes conflicting business interests through the whole process.

The attendees of the CLIM coalition meetings and those who contributed to CLIM along the way include all of those listed on the title page, but also include Bob Kerns, Chris Richardson, Paul Weineke and JonL White, and Kevin Males. Other contributors include Mike McMahon, Paul Robertson, John Hotchkiss, David Lowry, Mike Greenwald, Dave Schatsky, and Glenn Adams.

Sonya Keene, Ann Hathaway, and Ellen Golden all deserve credit for improving the quality of this specification (in part by authoring the original CLIM documentation), as does Mark Nahabedian.

Kent Pitman provided much useful advice on the mechanics of writing a specification, and this document has much improved because of that.

Last but not least are the people who have continued to use CLIM for the duration of this lengthy process. The people at BBN (Bruce Roberts and his group) have been a continuing source of valuable information and suggestions. Also, Markus Fischer of Symbolics GmbH had to deal with more than his fair share of problems, the result of using CLIM in novel ways.

Part I

Overview and Conventions

Chapter 1

Overview of CLIM

The Common Lisp Interface Manager (CLIM) is a powerful Lisp-based user interface programming interface that provides a layered set of portable facilities. These include basic windowing, input, and output services provided by Silica (which is a portable layer for constructing windows and toolkit objects); stream-oriented input and output extended with facilities such as output recording, presentations, and context sensitive input; and a compositional toolkit similar to those found in the X world that supports look and feel independence.

CLIM provides an API (applications programmer interface) to user interface facilities for the Lisp application programmer. CLIM does not compete with the window system or toolkits of the host machine (such as X/Motif or NEWS/OpenLook), but rather uses their services (to the extent that it makes sense) to integrate Lisp applications into the host's window environment. For example, CLIM "windows" are mapped onto one or more host windows, and input and output operations performed on the CLIM window are ultimately carried out by the host window system. CLIM will support a large number of host environments including Genera, X/Motif, X/OpenLook, the Macintosh, CLOE-386/486, and the Next machine.

The CLIM programmer is insulated from the complexities of portability, since the Lisp-based application need only deal with CLIM objects and functions regardless of their operating platform (that is, the combination of Lisp system, host computer, and host window environment). CLIM abstracts out many of the concepts common to all window environments. The programmer is encouraged to think in terms of these abstractions, rather than in the specific capabilities of a particular host system. For example, using CLIM, the application can specify the appearance of output in high-level terms and those high-level descriptions are turned into the appropriate appearance for the given host. Thus, the application has the same fundamental interface across multiple environments, although the details will differ from system to system.

Another important goal in the design and organization of CLIM is to provide a spectrum of user interface building options, all the way from detailed, low-level specification of "what goes where", to high-level user interface specification where the programmer leaves all of the details up to CLIM. This allows CLIM to balance the ease of use on one hand, and versatility on the other. The idea is that by using high level facilities, a programmer can build portable user interfaces quickly, whereas by recombining lower level facilities he can build his own programming and user interfaces according to his specific needs or requirements. For example, CLIM supports the development of applications independent of look and feel (also dubbed "chameleon look and feel") as well as the portable development of toolkit libraries that define and implement a particular look and feel.

In addition, CLIM's modular design allows application programs to exclude facilities that they do not use, and reimplement any or all of the substrate. To these ends, CLIM is specified and implemented in a layered, modular fashion. Each facility documented in this specification has several layers of interface, and each facility is independently specified and has a documented external interface.

The facilities provided by CLIM include:

Geometry CLIM provides provides geometric objects like point, rectangle, and transformations and functions for manipulating them.

Graphics CLIM provides a rich set of drawing functions, including ones for drawing complex geometric shapes, a wide variety of drawing options (such as line thickness), a sophisticated inking model, and color. CLIM provides full affine transforms, so that a drawing may be arbitrarily translated, rotated, and scaled (to the extent that the underlying window system supports the rendering of such objects).

Windowing CLIM provides a portable layer for implementing *sheet* classes (types of window-like objects) that are suited to support particular high level facilities or interfaces. The Silica module of CLIM defines a uniform interface for creating and managing hierarchies of these objects regardless of their sheet class. In addition, Silica defines standard sheet classes that define a straightforward window that is more or less equivalent in functionality to an X window.

Output Recording CLIM provides a facility for capturing all output done to a window. This facility provides the support for arbitrarily scrollable windows. In addition, this facility serves as the root for a variety of interesting high-level tools.

Formatted Output CLIM provides a set of high-level macros that enable programs to produce neatly formatted tabular and graphical displays with little or no effort.

Context Sensitive Input The *presentation type* facility of CLIM provides the ability to associate semantics with output, such that objects may be retrieved later via clicks on their displayed representation. This *sensitivity* comes along automatically and is integrated with the Common Lisp type system. A mechanism for type coercion is also included, providing the basis for powerful user interfaces.

Application Building CLIM provides a set of tools for organizing an application's top-level user interface and command processing loops centered on objects called frames. CLIM provides functionality for laying out frames under arbitrary constraints, managing command menus and/or menu bars, and linking user interface gestures with application commands. Using these tools, application writers can easily and quickly construct user interfaces that can grow flexibly from prototype to delivery.

Adaptive Toolkit CLIM provides a uniform interface to the standard compositional toolkits available in many environments. CLIM defines abstract panes that are analogous to the gadgets or widgets of a toolkit like InterViews or the X toolkit. CLIM fosters look and feel independence by specifying the interface of these abstract panes in terms of their function and not in terms of the details of their appearance or operation. If an application uses these interfaces, its user interface will adapt to use whatever toolkit is available in the host environment. By using this facility, application programmers can easily construct applications that will automatically conform to a variety of user interface standards. In addition, a portable lisp-based implementation of the abstract panes is provided.

Chapter 2

Conventions

This chapter describes the conventions used in this specification and in the CLIM software itself.

2.1 Audience, Goals, and Purpose

This document, the **CLIM Release 2 Specification**, is intended for vendors. While it does define the Application Programmer's Interface (API), that is, the functionality that a customer/consumer would use to write an application, it also defines the names and functionality of some internal parts of CLIM. These "portals" in implementation space allow one vendor to extend, for example, the output record mechanism and have it work with another vendor's implementation of incremental redisplay. We have attempted to carefully identify the appropriate "portals" so that the API can be implemented efficiently, but we have also tried not to overconstrain the specification so that it restricts creativity of implementation or the possibility for extension. This also affects the more sophisticated application writers who want to go a little below the published API but still want portable applications. This document defines which functionality is part of the advertised API, and which is part of the internal protocols.

In this document, we refer to three different audiences. A CLIM *user* is a person who uses an application program that was written using CLIM. A CLIM *programmer* is a person who writes application programs using CLIM. A CLIM *implementor* is a programmer who implements CLIM or extends it in some non-trivial way.

2.2 Package Structure

CLIM defines a variety of packages in order to provide its functionality. In general, no symbols except for the symbols in this specification should be added to those packages.

The `clim-lisp` package is intended to emulate ANSI-90 Common Lisp as much as possible, independent of the conformance of individual vendors. `clim-lisp` is the version of Common Lisp in which

CLIM is implemented and which the `clim-user` package uses instead of `common-lisp`. `clim-lisp` contains only exported symbols, and is locked in those implementations that allow package locking.

`clim` is the package where the symbols specified in this specification live. It contains only exported symbols and is locked in those implementations that allow package locking.

`clim-sys` is the package where useful "system-like" functionality lives, including such things as resources and multi-processing primitives. It contains concepts that are not part of Common Lisp, but which are not conceptually the province of CLIM itself. It contains only exported symbols and is locked in those implementations that allow package locking.

No code is written in any of the above packages, but rather code is written for symbols in the above packages. None of the above use any other packages (in the sense of the `:use` option to `defpackage`). A CLIM implementation might define a `clim-internals` package that uses each of the above packages, thus getting the definition of Lisp from `clim-lisp`. It would then implement the functionality of the symbols in `clim` and `clim-sys` in the `clim-internals` package.

`clim-user` is the package for programmers to use if they don't wish to create their own package. It is the CLIM analog of `common-lisp-user`.

2.3 "Spread" Point Arguments to Functions

Many functions that take point arguments come in two forms: *structured* and *spread*. Functions that take structured point arguments take the argument as a single `point` object. Functions that take spread point arguments take a pair of arguments that correspond to the x and y coordinates of the point.

Functions that take spread point arguments, or return spread point values have an asterisk in their name, for example, `draw-line*`.

2.4 Immutability of Objects

Most CLIM objects are *immutable*, that is, at the protocol level none of their components can be modified once the object is created. Examples of immutable objects include all of the members of the `region` classes, colors and opacities, text styles, and line styles. Since immutable objects by definition never change, functions in the CLIM API can safely capture immutable objects without first copying them. This also allows CLIM to cache immutable objects. Constructor functions that return immutable objects are free to either create and return a new object, or return an already existing object.

A few CLIM objects are *mutable*. Examples of mutable objects include streams and output records. Some components of mutable objects can be modified once the object has been created, usually via `setf` accessors.

In CLIM, object immutability is maintained at the class level. Throughout this specification, the immutability or mutability of a class will be explicitly specified.

Some immutable classes also allow *interning*. A class is said to be interning if it guarantees that two instances that are equivalent will always be `eq`. For example, if the class `color` were interning, calling `make-rgb-color` twice with the same arguments would return `eq` values. CLIM does not specify that any class is interning, however all immutable classes are allowed to be interning at the discretion of the implementation.

In some rare cases, CLIM will modify objects that are members of immutable classes. Such objects are referred to as being *volatile*. Extreme care must be take with volatile objects. This specification will note whenever some object that is part of the API is volatile.

2.5 Behavior of Interfaces

In this specification, any interfaces that take or return mutable objects can be classified in a few different ways.

Most functions *do not capture* their mutable input objects, that is, these functions will either not store the objects at all, or will copy any mutable objects before storing them, or perhaps store only some of the components of the objects. Later modifications to those objects will not affect the internal state of CLIM.

Some functions *may capture* their mutable input objects. That is, it is unspecified as to whether a CLIM implementation will or will not capture the mutable inputs to some function. For such functions, programmers should assume that these objects will be captured and should not modify these objects capriciously. Furthermore, it is unspecifed what will happen if these objects are later modified.

Some programmers might choose to create a mutable subclass of an immutable class. If CLIM captures an object that is a member of such a class, it is unspecified what will happen if the programmer later modifies that object. If a programmer passes such an object to a CLIM function that may capture its inputs, he is responsible for either first copying the object or ensuring that the object does not change later.

Some functions that return mutable objects are guaranteed to create *fresh outputs*. These objects can be modified without affecting the internal state of CLIM.

Functions that return mutable objects that are not fresh objects fall into two categories: those that return *read-only state*, and those that return *read/write state*. If a function returns read-only state, programmers must not modify that object; doing so might corrupt the state of CLIM. If a function returns read/write state, the modification of that object is part of CLIM's interface, and programmers are free to modify the object in ways that "make sense".

2.6 Protocol Classes and Predicates

CLIM supplies a set of predicates that can be called on an object to determine whether or not that object satisfies a certain protocol. These predicates can be implemented in one of two ways.

The first way is that a class implementing a particular protocol will inherit from a *protocol class* that corresponds to that protocol. A protocol class is an "abstract" class with no slots and no methods (except perhaps for some default methods), and exists only to indicate that some subclass obeys the protocol. In the case when a class inherits from a protocol class, the predicate could be implemented using `typep`. All of the CLIM geometry, region, sheet, and output record classes use this convention. For example, the presentation protocol class and predicate could be implemented in this way:

```
(defclass presentation () ())

(defun presentationp (object)
  (typep object 'presentation))
```

Note that in some implementations, it may be more efficient not to use `typep`, and instead use a generic function for the predicate. However, simply implementing a method for the predicate that returns *true* is not necessarily enough to assert that a class supports that protocol; the class must include the protocol class as a superclass.

CLIM always provides at least one "standard" instantiable class that implements each protocol.

The second way is that a class implementing a particular protocol must simply implement a method for a predicate generic function that returns *true* if and only if that class supports the protocol (otherwise, it returns *false*). Most of the CLIM stream classes use this convention. Protocol classes are not used in these cases because, as in the case of some of the stream classes, the underlying Lisp implementation may not be arranged so as to permit it. For example, the extended input stream protocol might be implemented in this way:

```
(defgeneric extended-input-stream-p (object))

(defmethod extended-input-stream-p ((object t)) nil)

(defmethod extended-input-stream-p ((object basic-extended-input-protocol)) t)

(defmethod extended-input-stream-p
           ((encapsulating-stream encapsulating-stream-mixin))
  (with-slots (stream) encapsulating-stream
    (extended-input-stream-p stream)))
```

Whenever a class inherits from a protocol class or returns *true* from the protocol predicate, the class must implement methods for all of the generic functions that make up the protocol.

2.7 Specialized Arguments to Generic Functions

Unless otherwise stated, this specification uses the following convention for specifying which arguments to generic functions are specialized:

- If the generic function is a `setf` function, the second argument is the one that is intended to be specialized.
- If the generic function is a "mapping" function (such as `map-over-region-set-regions`), the second argument (the object that specifies what is being mapped over) is the one that is specialized. The first argument (the functional argument) is not intended to be specialized.
- Otherwise, the first argument is the one that is intended to be specialized.

2.8 Sheet, Stream, or Medium Arguments to Macros

There are many macros that take a sheet, stream, or medium as one of the arguments, for example, `with-new-output-record` and `formatting-table`. In CLIM, this argument must be a variable bound to a sheet, stream, or medium; it may not be an arbitrary form that evaluates to a sheet, stream, or medium. `t` and sometimes `nil` are usually allowed as special cases; this causes the variable to be interpreted as a reference to another stream variable (usually `*standard-output*` for output macros, or `*query-io*` for input macros). Note that, while the variable outside the macro form and the variable inside the body share the same name, they cannot be assumed to be the same reference. That is, the macro is free to create a new binding for the variable. Thus, the following code fragment will not necessarily affect the value of *stream* outside the `formatting-table` form:

```
(formatting-table (stream)
  (setq stream some-other-stream)
  ...)
```

Furthermore, for the macros that take a sheet, stream, or medium argument, the position of that variable is always before any forms or other "inputs".

2.9 Macros that Expand into Calls to Advertised Functions

Some macros that take a "body" argument expand into a call to an advertised function that takes a functional argument. This functional argument will execute the suppled body. For a macro named "`with-`*environment*", the function is generally named "`invoke-with-`*environment*". For example, `with-drawing-options` might be defined as follows:

```
(defgeneric invoke-with-drawing-options (medium function &key)
  (declare (dynamic-extent function)))

(defmacro with-drawing-options ((medium &rest drawing-options) &body body)
  `(flet ((with-drawing-options-body (,medium) ,@body))
     (declare (dynamic-extent #'with-drawing-options-body))
     (invoke-with-drawing-options
       ,medium #'with-drawing-options-body ,@drawing-options)))
```

```
(defmethod invoke-with-drawing-options
           ((medium clx-display-medium) function &rest drawing-options)
  (with-drawing-options-merged-into-medium (medium drawing-options)
    (funcall function medium)))
```

2.10 Terminology Pertaining to Error Conditions

When this specification specifies that it "is an error" for some situation to occur, this means that:

- No valid CLIM program should cause this situation to occur.
- If this situation does occur, the effects and results are undefined as far as adherence to the CLIM specification is concerned.
- CLIM implementations are not required to detect such an error, although implementations are encouraged to provide such error detection whenever it is reasonable to do so.

When this specification specifies that some argument "must be a *type*" or uses the phrase "the *type argument*", this means that it is an error if the argument is not of the specified type. CLIM implementations are encouraged, but not required, to generate an argument type error for these situations.

When this specification says that "an error is signalled" in some situation, this means that:

- If the situation occurs, an error will be signalled using either `error` or `cerror`.
- Valid CLIM programs may rely on the fact that an error will be signalled.
- Every CLIM implementation is required to detect such an error.

When this specification says that "a condition is signalled" in some situation, this is just like "an error is signalled" with the exception that the condition will be signalled using `signal` instead of `error`.

Part II

Geometry Substrate

Chapter 3

Geometry

CLIM provides definitions for a variety of geometric objects, including points, lines, elliptical arcs, regions, and transformations. Both the graphics and windowing modules use the same set of geometric objects and functions. In this section, we describe regions, points, and the basic region classes. Transformations will be described in Chapter 5.

Most of these objects are described as if they are implemented using standard classes. However, this need not be the case. In particular, they may be implemented using structure classes, and some classes may exist only to name a place in the hierarchy—all members of such a class will be instances of that class's subclasses. The most important concern is that these classes must allow specializing generic functions.

The coordinate system in which the geometric objects reside is an abstract, continuous coordinate system. This abstract coordinate system is converted into "real world" coordinates only during operations such as rendering one of the objects on a display device.

Angles are measured in radians. Following standard conventions, when an angle is measured relative to a given line, a positive angle indicates an angle counter-clockwise from the line in the plane. When the angle from the positive x axis to the positive y axis is positive (that is, the positive y axis is counter-clockwise from the positive x axis), the coordinate system is said to be *right-handed.* When this angle is negative, the coordinate system is said to be *left-handed.* Thus, the cartesian coordinate system with x increasing to the right and y increasing upward is right-handed. A coordinate system with y increasing down is left-handed. (By default, CLIM streams are left handed, but no such default exists for sheets in general.)

3.1 General Regions

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

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

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

Another way to describe a region is that it maps every (x, y) pair into either *true* or *false* (meaning member or not a member, respectively, of the region). Later, in Chapter 14, we will generalize a region to something called a *design* that maps every point (x, y) into color and opacity values.

⇒ `region` [*Protocol Class*]

The protocol class that corresponds to a set of points. This includes both bounded and unbounded regions. If you want to create a new class that behaves like a region, it should be a subclass of `region`. Subclasses of `region` must obey the region protocol.

There is no general constructor called `make-region` because of the impossibility of a uniform way to specify the arguments to such a function.

⇒ `regionp` *object* [*Function*]

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

⇒ `path` [*Protocol Class*]

The protocol class `path` denotes bounded regions that have *dimensionality* 1 (that is, have length). It is a subclass of `region` and `bounding-rectangle`. If you want to create a new class that behaves like a path, it should be a subclass of `path`. Subclasses of `path` must obey the path protocol.

Constructing a `path` object with no length (via `make-line*`, for example) canonicalizes it to `+nowhere+`.

Some rendering models support the constructing of areas by filling a closed path. In this case, the path needs a direction associated with it. Since CLIM does not currently support the path-filling model, paths are directionless.

⇒ `pathp` *object* [*Function*]

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

⇒ `area` [*Protocol Class*]

The protocol class `area` denotes bounded regions that have dimensionality 2 (that is, have area). It is a subclass of `region` and `bounding-rectangle`. If you want to create a new class that behaves like an area, it should be a subclass of `area`. Subclasses of `area` must obey the area protocol.

Constructing an `area` object with no area (via `make-rectangle*`, for example) canonicalizes it to `+nowhere+`.

⇒ `areap` *object* [*Function*]

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

⇒ `+everywhere+` [*Constant*]
⇒ `+nowhere+` [*Constant*]

`+everywhere+` is the region that includes all the points on the two-dimensional infinite drawing plane. `+nowhere+` is the empty region, the opposite of `+everywhere+`.

3.1.1 The Region Predicate Protocol

The following generic functions comprise the region predicate protocol. All classes that are subclasses of `region` must either inherit or implement methods for these generic functions.

The methods for `region-equal`, `region-contains-region-p`, and `region-intersects-region-p` will typically specialize both the *region1* and *region2* arguments.

⇒ `region-equal` *region1 region2* [*Generic Function*]

Returns *true* if the two *regions* *region1* and *region2* contain exactly the same set of points, otherwise returns *false*.

⇒ `region-contains-region-p` *region1 region2* [*Generic Function*]

Returns *true* if all points in the *region* *region2* are members of the *region* *region1*, otherwise returns *false*.

⇒ `region-contains-point*-p` *region x y* [*Generic Function*]

Returns *true* if the point at (x, y) is contained in the *region* *region*, otherwise returns *false*. Since regions in CLIM are closed, this must return *true* if the point at (x, y) is on the region's boundary. CLIM implementations are permitted to return different non-`nil` values depending on whether the point is completely inside the region or is on the border.

`region-contains-point*-p` is a special case of `region-contains-region-p` in which the region is the point (x, y).

⇒ `region-intersects-region-p` *region1 region2* [*Generic Function*]

Returns *false* if `region-intersection` of the two *regions* *region1* and *region2* would be `+nowhere+`, otherwise returns *true*.

3.1.2 Region Composition Protocol

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

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

Some CLIM implementations might only implement a subset of full region composition. Because of the importance of rectangular regions and region sets that are the union of rectangular regions, every CLIM implementation is required to fully support all functions that use regions for those cases. (For example, CLIM implementations must be able do clipping and repainting on region sets composed entirely of axis-aligned rectangles.) If a CLIM implementation does not support some functions on non-rectangular region sets (for example, clipping), it must signal an error when an unsupported case is encountered; the exact details of this depend on the particular CLIM implementation.

⇒ `region-set` [*Protocol Class*]

The protocol class that represents a region set; a subclass of `region` and `bounding-rectangle`.

In addition to the three classes below, there may be other instantiable subclasses of `region-set` that represent special cases, for instance, some implementations might have a `standard-rectangle-set` class that represents the union of several axis-aligned rectangles.

Members of this class are immutable.

⇒ `region-set-p` *object* [*Function*]

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

⇒ `standard-region-union` [*Class*]
⇒ `standard-region-intersection` [*Class*]
⇒ `standard-region-difference` [*Class*]

These three instantiable classes respectively implement the union, intersection, and differences of regions. Implementations may, but are not required to, take advantage of the commutativity and associativity of union and intersection in order to "collapse" complicated region sets into simpler ones.

Region sets that are composed entirely of axis-aligned rectangles must be canonicalized into either a single rectangle or a union of rectangles. Furthermore, the rectangles in the union must not overlap each other.

The following generic functions comprise the region composition protocol. All classes that are subclasses of `region` must implement methods for these generic functions.

The methods for `region-union`, `region-intersection`, and `region-difference` will typically specialize both the *region1* and *region2* arguments.

⇒ `region-set-regions` *region* `&key` *normalize* [*Generic Function*]

Returns a sequence of the regions in the *region set region*. *region* can be either a *region set* or a "simple" region, in which case the result is simply a sequence of one element: *region*. This function returns objects

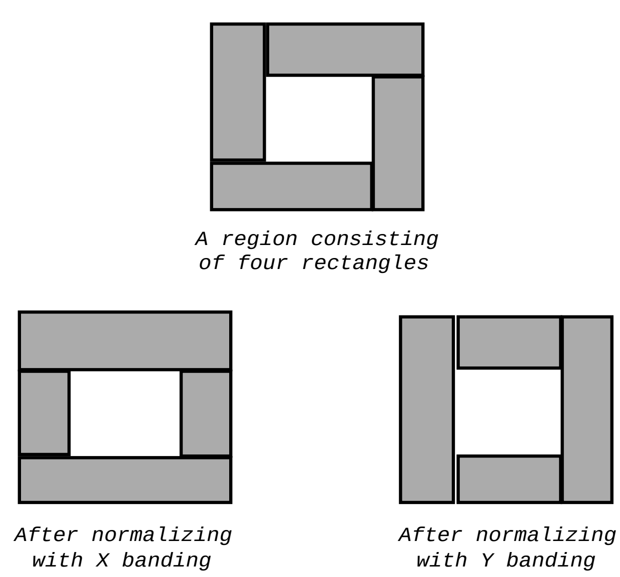


Figure 3.1: Normalization of rectangular region sets.

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

For the case of region sets that are unions of axis-aligned rectangles, the rectangles returned by `region-set-regions` are guaranteed not to overlap.

If *normalize* is supplied, it must be either `:x-banding` or `:y-banding`. If it is `:x-banding` and all the regions in *region* are axis-aligned rectangles, the result is normalized by merging adjacent rectangles with banding done in the x direction. If it is `:y-banding` and all the regions in *region* are rectangles, the result is normalized with banding done in the y direction. Normalizing a region set that is not composed entirely of axis-aligned rectangles using x- or y-banding causes CLIM to signal the `region-set-not-rectangular` error.

⇒ `map-over-region-set-regions` *function region* `&key` *normalize* [*Generic Function*]

Calls *function* on each region in the *region set region*. This is often more efficient than calling `region-set-regions`. *function* is a function of one argument, a region; it has dynamic extent. *region* can be either a *region set* or a "simple" region, in which case *function* is called once on *region* itself. *normalize* is as for `region-set-regions`.

⇒ `region-union` *region1 region2* [*Generic Function*]

Returns a region that contains all points that are in either of the *regions region1* or *region2* (possibly with some points removed in order to satisfy the dimensionality rule). The result of `region-union` always has dimensionality that is the maximum dimensionality of *region1* and *region2*. For example, the union of a path and an area produces an area; the union of two paths is a path.

`region-union` will return either a simple region or a member of the class `standard-region-union`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

⇒ `region-intersection` *region1 region2* [*Generic Function*]

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

`region-intersection` will return either a simple region or a member of the class `standard-region-intersection`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

⇒ `region-difference` *region1 region2* [*Generic Function*]

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

`region-difference` will return either a simple region or a member of the class `standard-region-difference`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

3.2 Other Region Types

The other types of regions are points, polylines, polygons, elliptical arcs, and ellipses. All of these region types are closed under affine transformations.

Issue: A reasonable way to address the perceived "bloating" caused by all of the following region types is to specify that CLIM must not internally use any region types except for points, rectangles, and region sets composed of rectangular regions. This allows CLIM implementations to omit support for the other region types from the "core" of CLIM, and include that support as a layered module. — SWM

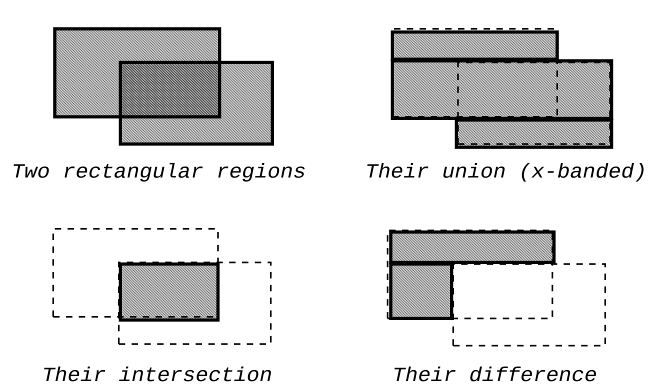


Figure 3.2: Examples of region union, intersection, and difference.

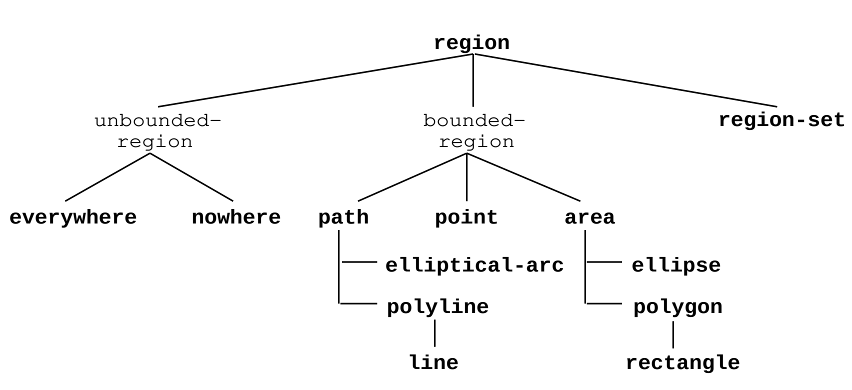


Figure 3.3: The class structure for all regions.

3.2.1 Points

A *point* is a mathematical point in the plane, designated by its coordinates, which are a pair of real numbers (where a real number is defined as either an integer, a ratio, or a floating point number). Points have neither area nor length (that is, they have dimensionality 0).

Note well that a point is *not* a pixel; CLIM models a drawing plane with continuous coordinates. This is discussed in more detail in Chapter 12.

⇒ `point` [*Protocol Class*]

The protocol class that corresponds to a mathematical point. This is a subclass of `region` and `bounding-rectangle`. If you want to create a new class that behaves like a point, it should be a subclass of `point`. Subclasses of `point` must obey the point protocol.

⇒ `pointp` *object* [*Function*]

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

⇒ `standard-point` [*Class*]

An instantiable class that implements a point. This is a subclass of `point`. This is the class that `make-point` instantiates. Members of this class are immutable.

⇒ `make-point` *x y* [*Function*]

Returns an object of class `standard-point` whose coordinates are *x* and *y*. *x* and *y* must be real numbers.

The Point Protocol

The following generic functions comprise the point API. Only `point-position*` is in the point protocol, that is, all classes that are subclasses of `point` must implement methods for `point-position*`, but need not implement methods for `point-x` and `point-y`.

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

Returns both the x and y coordinates of the point *point* as two values.

⇒ `point-x` *point* [*Generic Function*]
⇒ `point-y` *point* [*Generic Function*]

Returns the x or y coordinate of the *point point*, respectively. CLIM will supply default methods for `point-x` and `point-y` on the protocol class `point` that are implemented by calling `point-position*`.

3.2.2 Polygons and Polylines

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

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

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

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

⇒ `polyline` [*Protocol Class*]

The protocol class that corresponds to a polyline. This is a subclass of `path`. If you want to create a new class that behaves like a polyline, it should be a subclass of `polyline`. Subclasses of `polyline` must obey the polyline protocol.

⇒ `polylinep` *object* [*Function*]

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

⇒ `standard-polyline` [*Class*]

An instantiable class that implements a polyline. This is a subclass of `polyline`. This is the class that `make-polyline` and `make-polyline*` instantiate. Members of this class are immutable.

⇒ `make-polyline` *point-seq* `&key` *closed* [*Function*]
⇒ `make-polyline*` *coord-seq* `&key` *closed* [*Function*]

Returns an object of class `standard-polyline` consisting of the segments connecting each of the points in *point-seq* (or the points represented by the coordinate pairs in *coord-seq*). *point-seq* is a sequence of *points*; *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements.

If *closed* is *true*, then the segment connecting the first point and the last point is included in the polyline. The default for *closed* is *false*.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

⇒ `polygon` [*Protocol Class*]

The protocol class that corresponds to a mathematical polygon. This is a subclass of `area`. If you want to create a new class that behaves like a polygon, it should be a subclass of `polygon`. Subclasses of `polygon` must obey the polygon protocol.

⇒ `polygonp` *object* [*Function*]

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

⇒ `standard-polygon` [*Class*]

An instantiable class that implements a polygon. This is a subclass of `polygon`. This is the class that `make-polygon` and `make-polygon*` instantiate. Members of this class are immutable.

⇒ `make-polygon` *point-seq* [*Function*]
⇒ `make-polygon*` *coord-seq* [*Function*]

Returns an object of class `standard-polygon` consisting of the area contained in the boundary that is specified by the segments connecting each of the points in *point-seq* (or the points represented by the coordinate pairs in *coord-seq*). *point-seq* is a sequence of *points*; *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

The Polygon and Polyline Protocol

The following generic functions comprise the polygon and polyline protocol. All classes that are subclasses of either `polygon` or `polyline` must implement methods for these generic functions. Some of the functions below take an argument named *polygon-or-polyline*; this argument may be either a *polygon* or a *polyline*.

⇒ `polygon-points` *polygon-or-polyline* [*Generic Function*]

Returns a sequence of points that specify the segments in *polygon-or-polyline*. This function returns objects that reveal CLIM's internal state; do not modify those objects.

⇒ `map-over-polygon-coordinates` *function polygon-or-polyline* [*Generic Function*]

Applies *function* to all of the coordinates of the vertices of *polygon-or-polyline*. *function* is a function of two arguments, the x and y coordinates of the vertex; it has dynamic extent.

⇒ `map-over-polygon-segments` *function polygon-or-polyline* [*Generic Function*]

Applies *function* to the segments that compose *polygon-or-polyline*. *function* is a function of four arguments, the x and y coordinates of the start of the segment, and the x and y coordinates of the end of the segment; it has dynamic extent. When `map-over-polygon-segments` is called on a closed polyline, it will call *function* on the segment that connects the last point back to the first point.

⇒ `polyline-closed` *polyline* [*Generic Function*]

Returns *true* if the polyline *polyline* is closed, otherwise returns *false*. This function need be implemented only for *polylines*, not for *polygons*.

3.2.3 Lines

A line is a polyline consisting of a single segment.

⇒ `line` [*Protocol Class*]

The protocol class that corresponds to a mathematical line segment, that is, a polyline with only a single segment. This is a subclass of `polyline`. If you want to create a new class that behaves like a line, it should be a subclass of `line`. Subclasses of `line` must obey the line protocol.

⇒ `linep` *object* [*Function*]

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

⇒ `standard-line` [*Class*]

An instantiable class that implements a line segment. This is a subclass of `line`. This is the class that `make-line` and `make-line*` instantiate. Members of this class are immutable.

⇒ `make-line` *start-point end-point* [*Function*]
⇒ `make-line*` *start-x start-y end-x end-y* [*Function*]

Returns an object of class `standard-line` that connects the two *points start-point* and *end-point* (or the positions (*start-x*,*start-y*) and (*end-x*,*end-y*)).

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

The Line Protocol

The following generic functions comprise the line API. Only `line-start-point*` and `line-end-point*` are in the line protocol, that is, all classes that are subclasses of `line` must implement methods for `line-start-point*` and `line-end-point*`, but need not implement methods for `line-start-point` and `line-end-point`.

⇒ `line-start-point*` *line* [*Generic Function*]
⇒ `line-end-point*` *line* [*Generic Function*]

Returns the starting or ending point, respectively, of the *line line* as two real numbers representing the coordinates of the point.

⇒ `line-start-point` *line* [*Generic Function*]
⇒ `line-end-point` *line* [*Generic Function*]

Returns the starting or ending point of the *line line*, respectively.

CLIM will supply default methods for `line-start-point` and `line-end-point` on the protocol class `line` that are implemented by calling `line-start-point*` and `line-end-point*`.

3.2.4 Rectangles

Rectangles whose edges are parallel to the coordinate axes are a special case of polygon that can be specified completely by four real numbers (*x1*,*y1*,*x2*,*y2*). They are *not* closed under general affine transformations (although they are closed under rectilinear transformations).

⇒ `rectangle` [*Protocol Class*]

The protocol class that corresponds to a mathematical rectangle, that is, rectangular polygons whose sides are parallel to the coordinate axes. This is a subclass of `polygon`. If you want to create a new class that behaves like a rectangle, it should be a subclass of `rectangle`. Subclasses of `rectangle` must obey the rectangle protocol.

⇒ `rectanglep` *object* [*Function*]

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

⇒ `standard-rectangle` [*Class*]

An instantiable class that implements an axis-aligned rectangle. This is a subclass of `rectangle`. This is the class that `make-rectangle` and `make-rectangle*` instantiate. Members of this class are immutable.

⇒ `make-rectangle` *point1 point2* [*Function*]
⇒ `make-rectangle*` *x1 y1 x2 y2* [*Function*]

Returns an object of class `standard-rectangle` whose edges are parallel to the coordinate axes. One corner is at the *point point1* (or the position (*x1*,*y1*)) and the opposite corner is at the *point point2* (or the position (*x2*,*y2*)). There are no ordering constraints among *point1* and *point2* (or *x1* and *x2*, and *y1* and *y2*).

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

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

The Rectangle Protocol

The following generic functions comprise the rectangle API. Only `rectangle-edges*` is in the rectangle protocol, that is, all classes that are subclasses of `rectangle` must implement methods for `rectangle-edges*`, but need not implement methods for the remaining functions.

⇒ `rectangle-edges*` *rectangle* [*Generic Function*]

Returns the coordinates of the minimum x and y and maximum x and y of the rectangle *rectangle* as four values, *min-x*, *min-y*, *max-x*, and *max-y*.

⇒ `rectangle-min-point` *rectangle* [*Generic Function*]
⇒ `rectangle-max-point` *rectangle* [*Generic Function*]

Returns the min point and max point of the *rectangle rectangle*, respectively. The position of a rectangle is specified by its min point.

CLIM will supply default methods for `rectangle-min-point` and `rectangle-max-point` on the protocol class `rectangle` that are implemented by calling `rectangle-edges*`.

⇒ `rectangle-min-x` *rectangle* [*Generic Function*]
⇒ `rectangle-min-y` *rectangle* [*Generic Function*]
⇒ `rectangle-max-x` *rectangle* [*Generic Function*]
⇒ `rectangle-max-y` *rectangle* [*Generic Function*]

Returns (respectively) the minimum x and y coordinate and maximum x and y coordinate of the *rectangle rectangle*.

CLIM will supply default methods for these four generic functions on the protocol class `rectangle` that are implemented by calling `rectangle-edges*`.

⇒ `rectangle-width` *rectangle* [*Generic Function*]
⇒ `rectangle-height` *rectangle* [*Generic Function*]
⇒ `rectangle-size` *rectangle* [*Generic Function*]

`rectangle-width` returns the width of the *rectangle rectangle*, which is the difference between the maximum x and its minimum x. `rectangle-height` returns the height, which is the difference between the maximum y and its minimum y. `rectangle-size` returns two values, the width and the height.

CLIM will supply default methods for these four generic functions on the protocol class `rectangle` that are implemented by calling `rectangle-edges*`.

3.2.5 Ellipses and Elliptical Arcs

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

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

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

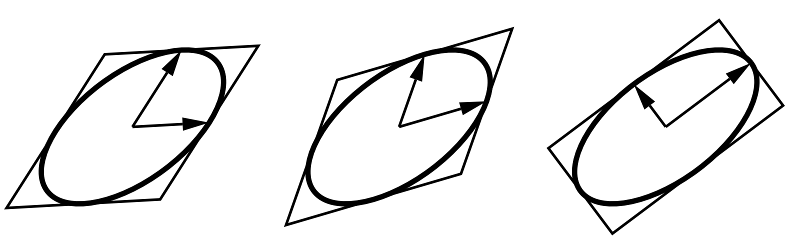


Figure 3.4: Different vectors may specify the same ellipse.

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

Note that several different parallelograms specify the same ellipse. One parallelogram is bound to be a rectangle—the vectors will be perpendicular and correspond to the semi-axes of the ellipse.

The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting $dx_2 = dy_1 = 0$ or $dx_1 = dy_2 = 0$.

⇒ `ellipse` [*Protocol Class*]

The protocol class that corresponds to a mathematical ellipse. This is a subclass of `area`. If you want to create a new class that behaves like an ellipse, it should be a subclass of `ellipse`. Subclasses of `ellipse` must obey the ellipse protocol.

⇒ `ellipsep` *object* [*Function*]

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

⇒ `standard-ellipse` [*Class*]

An instantiable class that implements an ellipse. This is a subclass of `ellipse`. This is the class that `make-ellipse` and `make-ellipse*` instantiate. Members of this class are immutable.

⇒ `make-ellipse` *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *start-angle end-angle* [*Function*]

⇒ `make-ellipse*` *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *start-angle end-angle* [*Function*]

Returns an object of class `standard-ellipse`. The center of the ellipse is at the *point* *center-point* (or the position (*center-x*,*center-y*)).

Two vectors, (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*) specify the bounding parallelogram of the ellipse as explained above. All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the `ellipse-not-well-defined` error will be signalled. The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

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

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

⇒ `elliptical-arc` [*Protocol Class*]

The protocol class that corresponds to a mathematical elliptical arc. This is a subclass of `path`. If you want to create a new class that behaves like an elliptical arc, it should be a subclass of `elliptical-arc`. Subclasses of `elliptical-arc` must obey the elliptical arc protocol.

⇒ `elliptical-arc-p` *object* [*Function*]

Returns *true* if *object* is an *elliptical arc*, otherwise returns *false*.

⇒ `standard-elliptical-arc` [*Class*]

An instantiable class that implements an elliptical arc. This is a subclass of `elliptical-arc`. This is the class that `make-elliptical-arc` and `make-elliptical-arc*` instantiate. Members of this class are immutable.

⇒ `make-elliptical-arc` *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *start-angle end-angle* [*Function*]
⇒ `make-elliptical-arc*` *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *start-angle end-angle* [*Function*]

Returns an object of class `standard-elliptical-arc`. The center of the ellipse is at the *point* *center-point* (or the position (*center-x*,*center-y*)).

Two vectors, (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*), specify the bounding parallelogram of the ellipse as explained above. All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the `ellipse-not-well-defined` error will be signalled. The special case of an elliptical arc with its axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

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

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

The Ellipse and Elliptical Arc Protocol

The following functions apply to both ellipses and elliptical arcs. In all cases, the name *elliptical-object* means that the argument may be an *ellipse* or an *elliptical arc*. These generic functions comprise the ellipse protocol. All classes that are subclasses of either `ellipse` or `elliptical-arc` must implement methods for these functions.

⇒ `ellipse-center-point*` *elliptical-object* [*Generic Function*]

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

⇒ `ellipse-center-point` *elliptical-object* [*Generic Function*]

Returns the center point of *elliptical-object*.

`ellipse-center-point` is part of the ellipse API, but not part of the ellipse protocol. CLIM will supply default methods for `ellipse-center-point` on the protocol classes `ellipse` and `elliptical-arc` that are implemented by calling `ellipse-center-point*`.

⇒ `ellipse-radii` *elliptical-object* [*Generic Function*]

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

⇒ `ellipse-start-angle` *elliptical-object* [*Generic Function*]

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

⇒ `ellipse-end-angle` *elliptical-object* [*Generic Function*]

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

Chapter 4

Bounding Rectangles

4.1 Bounding Rectangles

Every bounded region has a derived *bounding rectangle*, which is a rectangular region whose sides are parallel to the coordinate axes. Therefore, every bounded region participates in the bounding rectangle protocol. The bounding rectangle for a region is the smallest rectangle that contains every point in the region. However, the bounding rectangle may contain additional points as well. Unbounded regions do not have a bounding rectangle and do not participate in the bounding rectangle protocol. Other objects besides bounded regions participate in the bounding rectangle protocol, such as sheets and output records.

The coordinate system in which the bounding rectangle is maintained depends on the context. For example, the coordinates of the bounding rectangle of a sheet are expressed in the sheet's parent's coordinate system. For output records, the coordinates of the bounding rectangle are maintained in the coordinate system of the stream with which the output record is associated.

Note that the bounding rectangle of a transformed region is not in general the same as the result of transforming the bounding rectangle of a region, as shown in Figure 4.1. For transformations that satisfy `rectilinear-transformation-p`, the following equality holds. For all other transformations, it does not hold.

```
(region-equal
  (transform-region transformation (bounding-rectangle region))
  (bounding-rectangle (transform-region transformation region)))
```

CLIM uses bounding rectangles for a variety of purposes. For example, repainting of windows is driven from the bounding rectangle of the window's viewport, intersected with a "damage" region. The formatting engines used by `formatting-table` and `formatting-graph` operate on the bounding rectangles of the output records in the output. Bounding rectangles are also used internally by CLIM to achieve greater efficiency. For instance, when performing hit detection to see if the pointer is within the region of an output record, CLIM first checks to see if the pointer is within the bounding rectangle of the output record.

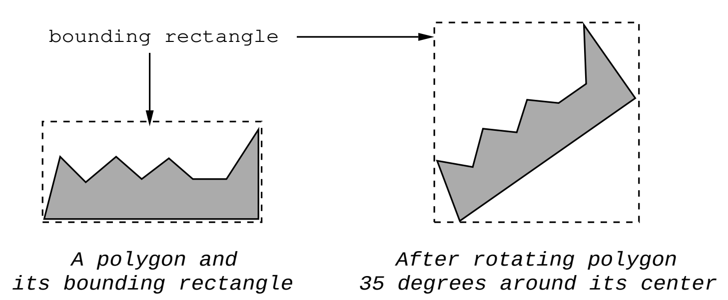


Figure 4.1: The bounding rectangle of an output record.

Note that the bounding rectangle for an output record may have a different size depending on the medium on which the output record is rendered. Consider the case of rendering text on different output devices; the font chosen for a particular text style may vary considerably in size from one device to another.

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

The protocol class that represents a bounding rectangle. If you want to create a new class that behaves like a bounding rectangle, it should be a subclass of `bounding-rectangle`. Subclasses of `bounding-rectangle` must obey the bounding rectangle protocol.

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

Bounding rectangles are immutable, but since they reflect the live state of such mutable objects as sheets and output records, bounding rectangles are volatile. Therefore, programmers must not depend on the bounding rectangle associated with a mutable object remaining constant.

⇒ `bounding-rectangle-p` *object* [*Function*]

Returns *true* if *object* is a *bounding rectangle* (that is, supports the bounding rectangle protocol), otherwise returns *false*.

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

An instantiable class that implements a bounding rectangle. This is a subclass of both `bounding-rectangle` and `rectangle`, that is, standard bounding rectangles obey the rectangle protocol.

`make-bounding-rectangle` returns an object of this class.

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

⇒ `make-bounding-rectangle` *x1 y1 x2 y2* [*Function*]

Returns an object of the class `standard-bounding-rectangle` with the edges specified by *x1*, *y1*, *x2*, and *y2*, which must be real numbers.

x1, *y1*, *x2*, and *y2* are "canonicalized" in the following way. The min point of the rectangle has an x coordinate that is the smaller of *x1* and *x2* and a y coordinate that is the smaller of *y1* and *y2*. The max point of the rectangle has an x coordinate that is the larger of *x1* and *x2* and a y coordinate that is the larger of *y1* and *y2*. (Therefore, in a right-handed coordinate system the canonicalized values of *x1*, *y1*, *x2*, and *y2* correspond to the left, top, right, and bottom edges of the rectangle, respectively.)

This function returns fresh objects that may be modified.

4.1.1 The Bounding Rectangle Protocol

The following generic function comprises the bounding rectangle protocol. All classes that participate in this protocol (including all subclasses of `region` that are bounded regions) must implement a method for `bounding-rectangle*`.

⇒ `bounding-rectangle*` *region* [*Generic Function*]

Returns the bounding rectangle of *region* as four real numbers specifying the x and y coordinates of the min point and the x and y coordinates of the max point of the rectangle. The argument *region* must be either a bounded region (such as a line or an ellipse) or some other object that obeys the bounding rectangle protocol, such as a sheet or an output record.

The four returned values *min-x*, *min-y*, *max-x*, and *max-y* will satisfy the inequalities

$$minx \leq maxx$$
$$miny \leq maxy$$

⇒ `bounding-rectangle` *region* [*Generic Function*]

Returns the bounding rectangle of *region* as an object that is a subclass of `rectangle` (described in Section 3.2.4). The argument *region* must be either a bounded region (such as a line or an ellipse) or some other object that obeys the bounding rectangle protocol, such as a sheet or an output record.

It is unspecified whether `bounding-rectangle` will or will not create a new object each time it is called. Many CLIM implementations will cache the bounding rectangle for sheets and output records. The implication of this is that, since bounding rectangles are volatile, programmers should depend on the object returned by `bounding-rectangle` remaining constant.

`bounding-rectangle` is part of the bounding rectangle API, but not part of the bounding rectangle protocol. CLIM will supply a default method for `bounding-rectangle` on the protocol class `bounding-rectangle` that is implemented by calling `bounding-rectangle*`.

4.1.2 Bounding Rectangle Convenience Functions

The functions described below are part of the bounding rectangle API, but are not part of the bounding rectangle protocol. They are provided as a convenience to programmers who wish to specialize classes that participate in the bounding rectangle protocol, but do not complicate the task of those programmers who define their own types (such as sheet classes) that participate in this protocol.

CLIM will supply default methods for all of these generic functions on the protocol class `bounding-rectangle` that are implemented by calling `bounding-rectangle*`.

⇒ `with-bounding-rectangle*` *(min-x min-y max-x max-y) region* `&body` *body* [*Macro*]

Binds *min-x*, *min-y*, *max-x*, and *max-y* to the edges of the bounding rectangle of *region*, and then executes *body* in that context. The argument *region* must be either a bounded region (such as a line or an ellipse) or some other object that obeys the bounding rectangle protocol, such as a sheet or an output record.

The arguments *min-x*, *min-y*, *max-x*, and *max-y* are not evaluated. *body* may have zero or more declarations as its first forms.

`with-bounding-rectangle*` must be implemented by calling `bounding-rectangle*`.

⇒ `bounding-rectangle-position*` *region* [*Generic Function*]

Returns the position of the bounding rectangle of *region*. The position of a bounding rectangle is specified by its min point.

⇒ `bounding-rectangle-min-x` *region* [*Generic Function*]
⇒ `bounding-rectangle-min-y` *region* [*Generic Function*]
⇒ `bounding-rectangle-max-x` *region* [*Generic Function*]
⇒ `bounding-rectangle-max-y` *region* [*Generic Function*]

Returns (respectively) the x and y coordinates of the min point and the x and y coordinate of the max point of the bounding rectangle of *region*. The argument *region* must be either a bounded region or some other object that obeys the bounding rectangle protocol.

⇒ `bounding-rectangle-width` *region* [*Generic Function*]
⇒ `bounding-rectangle-height` *region* [*Generic Function*]
⇒ `bounding-rectangle-size` *region* [*Generic Function*]

Returns the width, height, or size (as two values, the width and height) of the bounding rectangle of *region*, respectively. The argument *region* must be either a bounded region or some other object that obeys the bounding rectangle protocol.

The width of a bounding rectangle is the difference between its maximum x coordinate and its minimum

x coordinate. The height is the difference between the maximum y coordinate and its minimum y coordinate.

Chapter 5

Affine Transformations

An *affine transformation* is a mapping from one coordinate system onto another that preserves straight lines. In other words, if you take a number of points that fall on a straight line and apply an affine transformation to their coordinates, the transformed coordinates will describe a straight line in the new coordinate system. General affine transformations include all the sorts of transformations that CLIM uses, namely, translations, scaling, rotations, and reflections.

5.1 Transformations

⇒ `transformation` [*Protocol Class*]

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

All of the instantiable transformation classes provided by CLIM are immutable.

⇒ `transformationp` *object* [*Function*]

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

⇒ `+identity-transformation+` [*Constant*]

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

5.1.1 Transformation Conditions

⇒ `transformation-error` [*Condition*]

The class that is the superclass of the following three conditions. This class is a subclass of `error`.

⇒ `transformation-underspecified` [*Condition*]

The error that is signalled when `make-3-point-transformation` is given three collinear image points.

⇒ `reflection-underspecified` [*Condition*]

The error that is signalled when `make-reflection-transformation` is given two coincident points.

⇒ `singular-transformation` [*Condition*]

The error that is signalled when `invert-transformation` is called on a singular transformation, that is, a transformation that has no inverse.

5.2 Transformation Constructors

The following transformation constructors do not capture any of their inputs. The constructors all create objects that are subclasses of `transformation`.

⇒ `make-translation-transformation` *translation-x translation-y* [*Function*]

A translation is a transformation that preserves length, angle, and orientation of all geometric entities.

`make-translation-transformation` returns a transformation that translates all points by *translation-x* in the x direction and *translation-y* in the y direction. *translation-x* and *translation-y* must be real numbers.

⇒ `make-rotation-transformation` *angle* `&optional` *origin* [*Function*]
⇒ `make-rotation-transformation*` *angle* `&optional` *origin-x origin-y* [*Function*]

A rotation is a transformation that preserves length and angles of all geometric entities. Rotations also preserve one point (the origin) and the distance of all entities from that point.

`make-rotation-transformation` returns a transformation that rotates all points by *angle* (which is a real number indicating an angle in radians) around the point *origin*. If *origin* is supplied it must be a point; if not supplied it defaults to $(0, 0)$. *origin-x* and *origin-y* must be real numbers.

⇒ `make-scaling-transformation` *scale-x scale-y* `&optional` *origin* [*Function*]
⇒ `make-scaling-transformation*` *scale-x scale-y* `&optional` *origin-x origin-y* [*Function*]

There is no single definition of a scaling transformation. Transformations that preserve all angles and multiply all lengths by the same factor (preserving the "shape" of all entities) are certainly scaling transformations. However, scaling is also used to refer to transformations that scale distances in the x direction by one amount and distances in the y direction by another amount.

`make-scaling-transformation` returns a transformation that multiplies the x-coordinate distance of every point from *origin* by *scale-x* and the y-coordinate distance of every point from *origin* by *scale-y*. *scale-x* and *scale-y* must be real numbers. If *origin* is supplied it must be a point; if not supplied it defaults to $(0, 0)$. *origin-x* and *origin-y* must be real numbers.

⇒ `make-reflection-transformation` *point1 point2* [*Function*]
⇒ `make-reflection-transformation*` *x1 y1 x2 y2* [*Function*]

A reflection is a transformation that preserves lengths and magnitudes of angles, but changes the sign (or "handedness") of angles. If you think of the drawing plane on a transparent sheet of paper, a reflection is a transformation that "turns the paper over".

`make-reflection-transformation` returns a transformation that reflects every point through the line passing through the *points point1* and *point2* (or through the positions $(x1, y1)$ and $(x2, y2)$ in the case of the spread version).

⇒ `make-transformation` *mxx mxy myx myy tx ty* [*Function*]

Returns a general transformation whose effect is:

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

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

All of the arguments to `make-transformation` must be real numbers.

⇒ `make-3-point-transformation` *point-1 point-2 point-3 point-1-image point-2-image point-3-image* [*Function*]

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

If *point-1*, *point-2* and *point-3* are collinear, the `transformation-underspecified` error will be signalled. If *point-1-image*, *point-2-image* and *point-3-image* are collinear, the resulting transformation will be singular (that is, will have no inverse) but this is not an error.

⇒ `make-3-point-transformation*` *x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image* [*Function*]

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

If the positions $(x1, y1)$, $(x2, y2)$ and $(x3, y3)$ are collinear, the `transformation-underspecified` error will be signalled. If (*x1-image*,*y1-image*), (*x2-image*,*y2-image*), and (*x3-image*,*y3-image*) are collinear, the resulting transformation will be singular but this is not an error.

This is the spread version of `make-3-point-transformation`.

5.3 The Transformation Protocol

The following subsections describe the transformation protocol. All classes that are subclasses of `transformation` must implement methods for all of the generic functions in the following subsections.

5.3.1 Transformation Predicates

In all of the functions below, the argument named *transformation* must be a transformation.

⇒ `transformation-equal` *transformation1 transformation2* [*Generic Function*]

Returns *true* if the two *transformations transformation1* and *transformation2* have equivalent effects (that is, are mathematically equal), otherwise returns *false*.

Implementations are encouraged to allow transformations that are not numerically equal due to floating-point roundoff errors to be `transformation-equal`. An appropriate level of “fuzziness” is `single-float-epsilon`, or some small multiple of `single-float-epsilon`.

⇒ `identity-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* is equal (in the sense of `transformation-equal`) to the identity transformation, otherwise returns *false*.

⇒ `invertible-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* has an inverse, otherwise returns *false*.

⇒ `translation-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* is a pure translation, that is, a transformation such that there are two distance components dx and dy and every point (x, y) is moved to $(x + dx, y + dy)$. Otherwise, `translation-transformation-p` returns *false*.

⇒ `reflection-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* inverts the “handedness” of the coordinate system, otherwise returns *false*. Note that this is a very inclusive category—transformations are considered reflections even if they distort, scale, or skew the coordinate system, as long as they invert the handedness.

Figure 5.1: The predicates for analyzing the mathematical properties of a transformation.

⇒ `rigid-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* transforms the coordinate system as a rigid object, that is, as a combination of translations, rotations, and pure reflections. Otherwise, it returns *false*.

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

⇒ `even-scaling-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* multiplies all x lengths and y lengths by the same magnitude, otherwise returns *false*. It does include pure reflections through vertical and horizontal lines.

⇒ `scaling-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* multiplies all x lengths by one magnitude and all y lengths by another magnitude, otherwise returns *false*. This category includes even scalings as a subset.

⇒ `rectilinear-transformation-p` *transformation* [*Generic Function*]

Returns *true* if the *transformation transformation* will always transform any axis-aligned rectangle into another axis-aligned rectangle, otherwise returns *false*. This category includes scalings as a subset, and also includes 90 degree rotations.

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

5.3.2 Composition of Transformations

If we transform from one coordinate system to another, then from the second to a third coordinate system, we can regard the resulting transformation as a single transformation resulting from *composing* the two component transformations. It is an important and useful property of affine transformations that they are closed under composition. Note that composition is not commutative; in general, the result of applying transformation A and then applying transformation B is not the same as applying B first, then A.

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

⇒ `compose-transformations` *transformation1 transformation2* [*Generic Function*]

Returns a transformation that is the mathematical composition of its arguments. Composition is in right-to-left order, that is, the resulting transformation represents the effects of applying the *transformation transformation2* followed by the *transformation transformation1*.

⇒ `invert-transformation` *transformation* [*Generic Function*]

Returns a transformation that is the inverse of the transformation *transformation*. The result of composing a transformation with its inverse is equal to the identity transformation.

If *transformation* is singular, `invert-transformation` will signal the `singular-transformation` error, with a named restart that is invoked with a transformation and makes `invert-transformation` return that transformation. This is to allow a drawing application, for example, to use a generalized inverse to transform a region through a singular transformation.

Note that with finite-precision arithmetic there are several low-level conditions that might occur during the attempt to invert a singular or "almost singular" transformation. (These include computation of a zero determinant, floating-point underflow during computation of the determinant, or floating-point overflow during subsequent multiplication.) `invert-transformation` must signal the `singular-transformation` error for all of these cases.

⇒ `compose-translation-with-transformation` *transformation dx dy* [*Function*]
⇒ `compose-scaling-with-transformation` *transformation sx sy* `&optional` *origin* [*Function*]

⇒ `compose-rotation-with-transformation` *transformation angle* `&optional` *origin* [*Function*]

These functions create a new transformation by composing the transformation *transformation* with a given translation, scaling, or rotation, respectively. The order of composition is that the translation, scaling, or rotation "transformation" is first, followed by *transformation*.

dx and *dy* are as for `make-translation-transformation`. *sx* and *sy* are as for `make-scaling-transformation`. *angle* and *origin* are as for `make-rotation-transformation`.

Note that these functions could be implemented by using the various constructors and `compose-transformations`. They are provided, because it is common to build up a transformation as a series of simple transformations.

⇒ `compose-transformation-with-translation` *transformation dx dy* [*Function*]
⇒ `compose-transformation-with-scaling` *transformation sx sy* `&optional` *origin* [*Function*]

⇒ `compose-transformation-with-rotation` *transformation angle* `&optional` *origin* [*Function*]

These functions create a new transformation by composing a given translation, scaling, or rotation, respectively, with the transformation *transformation*. The order of composition is *transformation* is first, followed by the translation, scaling, or rotation "transformation".

dx and *dy* are as for `make-translation-transformation`. *sx* and *sy* are as for `make-scaling-transformation`. *angle* and *origin* are as for `make-rotation-transformation`.

Note that these functions could be implemented by using the various constructors and `compose-transformations`. They are provided, because it is common to build up a transformation as a series of simple transformations.

5.3.3 Applying Transformations

Transforming a region applies a coordinate transformation to that region, thus moving its position on the drawing plane, rotating it, or scaling it. Note that transforming a region does not side-effect the *region* argument; it is free to either create a new region or return an existing (cached) region.

These generic functions must be implemented for all classes of transformations. Furthermore, all subclasses of `region` and `design` must implement methods for `transform-region` and `untransform-region`. That is, methods for the following generic functions will typically specialize both the *transformation* and region arguments.

⇒ `transform-region` *transformation region* [*Generic Function*]

Applies *transformation* to the region *region*, and returns the transformed region.

⇒ `untransform-region` *transformation region* [*Generic Function*]

This is exactly equivalent to
`(transform-region (invert-transformation` *transformation*`)` *region*`)` .

CLIM provides a default method for `untransform-region` on the `transformation` protocol class that does exactly this.

⇒ `transform-point*` *transformation x y* [*Generic Function*]

Applies the transformation *transformation* to the point whose coordinates are the real numbers *x* and *y*, and returns two values, the transformed x coordinate and the transformed y coordinate.

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

⇒ `untransform-point*` *transformation x y* [*Generic Function*]

This is exactly equivalent to
`(transform-point* (invert-transformation` *transformation*`)` *x y*`)` .

CLIM provides a default method for `untransform-point*` on the `transformation` protocol class that does exactly this.

⇒ `transform-distance` *transformation dx dy* [*Generic Function*]

Applies the transformation *transformation* to the distance represented by the real numbers *dx* and *dy*, and returns two values, the transformed *dx* and the transformed *dy*.

A distance represents the difference between two points. It does *not* transform like a point.

⇒ `untransform-distance` *transformation dx dy* [*Generic Function*]

This is exactly equivalent to
`(transform-distance (invert-transformation` *transformation*`)` *dx dy*`)` .

CLIM provides a default method for `untransform-distance` on the `transformation` protocol class that does exactly this.

⇒ `transform-rectangle*` *transformation x1 y1 x2 y2* [*Generic Function*]

Applies the transformation *transformation* to the rectangle specified by the four coordinate arguments, which are real numbers. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for `make-bounding-rectangle`. Returns four values that specify the minimum and maximum points of the transformed rectangle in the order *min-x*, *min-y*, *max-x*, and *max-y*.

It is an error is *transformation* does not satisfy `rectilinear-transformation-p`.

`transform-rectangle*` is the spread version of `transform-region` in the case where the transformation is rectilinear and the region is a rectangle.

⇒ `untransform-rectangle*` *transformation x1 y1 x2 y2* [*Generic Function*]

This is exactly equivalent to
`(transform-rectangle* (invert-transformation` *transformation*`)` *x1 y1 x2 y2*`)` .

CLIM provides a default method for `untransform-rectangle*` on the `transformation` protocol class that does exactly this.

Part III

Windowing Substrate

Chapter 6

Overview of Window Facilities

6.1 Introduction

A central notion in organizing graphical user interfaces is allocating screen regions to particular tasks and recursively subdividing these regions into subregions. The layer of CLIM known as Silica defines an extensible framework for constructing, using, and managing such *hierarchies of interactive regions.* This framework allows uniform treatment of the following things:

- Window objects like those in X or NeWS.
- Lightweight gadgets typical of toolkit layers.
- Structured graphics like output records and an application's presentation objects.
- Objects that act as Lisp handles for windows or gadgets implemented in a different language (such as OpenLook gadgets implemented in C).

From the perspective of most CLIM users, Silica plays the role of a window system. However Silica will usually utilize the services of a window system platform to provide efficient windowing, input, and output facilities. In this specification, such window system platforms will be referred to as host window or display servers.

The fundamental abstraction defined by Silica is called a *sheet.* A sheet can participate in a relationship called a *windowing relationship.* This relationship is one in which one sheet called the *parent* provides space to a number of other sheets called *children.* Support for establishing and maintaining this kind of relationship is the essence of what window systems provide. At any point in time, Silica allows a sheet to be a child in one relationship called its *youth windowing relationship* and a parent in another relationship called its *adult windowing relationship.*

Programmers can manipulate unrooted hierarchies of sheets (those without a connection to any particular display server). However, a sheet hierarchy must be attached to a display server to make it visible. *Ports* and *grafts* provide the functionality for managing this capability. A *port* is an abstract connection to a display service that is responsible for managing host display server resources and for processing input events received from the host display server. A *graft* is a special kind of sheet that represents a host window, typically a root window (that is, a screen-level window). A sheet is attached to a display by making it a child of a graft, which represents an appropriate host window. The sheet will then appear to be a child of that host window. So, a sheet is put onto a particular screen by making it a child of an appropriate graft and enabling it. Ports and grafts are described in detail in Chapter 9.

Issue: An open issue in Silica's design is allowing a sheet to participate in multiple youth relationships. — RSL

Issue: Allowing a sheet to have multiple parents at different times is necessary for allowing the same pane to appear in multiple layouts of a frame. We need to resolve this. — SWM

6.2 Properties of Sheets

Sheets have the following properties:

A coordinate system Provides the ability to refer to locations in a sheet's abstract plane.

A region Defines an area within a sheet's coordinate system that indicates the area of interest within the plane, that is, a clipping region for output and input. This typically corresponds to the visible region of the sheet on the display.

A parent A sheet that is the parent in a windowing relationship in which this sheet is a child.

Children An ordered set of sheets that are each a child in a windowing relationship in which this sheet is a parent. The ordering of the set corresponds to the stacking order of the sheets. Not all sheets have children.

A transformation Determines how points in this sheet's coordinate system are mapped into points in its parents.

An enabled flag Indicates whether the sheet is currently actively participating in the windowing relationship with its parent and siblings.

An event handler A procedure invoked when the display server wishes to inform CLIM of external events.

Output state A set of values used when CLIM causes graphical or textual output to appear on the display. This state is often represented by a medium.

6.3 Sheet Protocols

A sheet is a participant in a number of protocols. Every sheet must provide methods for the generic functions that make up these protocols. These protocols are:

The windowing protocol Describes the relationships between the sheet and its parent and children (and, by extension, all of its ancestors and descendants).

The input protocol Provides the event handler for a sheet. Events may be handled synchronously, asynchronously, or not at all.

The output protocol Provides graphical and textual output, and manages descriptive output state such as color, transformation, and clipping.

The repaint protocol Invoked by the event handler and by user programs to ensure that the output appearing on the display device appears as the program expects it to appear.

The notification protocol Invoked by the event handler and user programs to ensure that CLIM's representation of window system information is equivalent to the display server's.

These protocols may be handled directly by a sheet, queued for later processing by some other agent, or passed on to a delegate sheet for further processing.

6.4 Sample Silica Program

```
(defun test-silica ()
  (let* ((sheet
          (make-standard-sheet :parent (find-graft)
                               :transformation (make-translation 100 100)
                               :region (make-rectangle 0 0 200 200 )))
         (w (bounding-rectangle-width sheet))
         (h (bounding-rectangle-height sheet))
         (x1 (/ w 4))
         (y1 (/ h 4))
         (x2 (* 3 (/ w 4)))
         (y2 (* 3 (/ h 4))))
    (enable-sheet sheet t)
    (draw-line sheet (make-point x1 y1) (make-point x1 y2))
    (draw-line sheet (make-point x1 y2) (make-point x2 y2))
    (draw-line sheet (make-point x2 y2) (make-point x2 y1))
    (draw-line sheet (make-point x2 y1) (make-point x1 y1))
    (draw-line* sheet x1 y1 x2 y2)
    (draw-line* sheet x1 y2 x2 y1)
    (output-finish sheet)
    (sleep 1)
    (disable-sheet sheet)
    (sheet-disown-child (sheet-parent sheet) sheet)))
```

Issue: This needs to be a set of more compelling examples. — York

Chapter 7

Properties of Sheets

⇒ `sheet` [*Protocol Class*]

The protocol class that corresponds to a sheet, a subclass of `bounding-rectangle`. This and the next chapter describe all of the sheet protocols. If you want to create a new class that behaves like a sheet, it should be a subclass of `sheet`. Subclasses of `sheet` must obey the sheet protocol.

All of the subclasses of `sheet` are mutable.

⇒ `sheetp` *object* [*Function*]

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

7.1 Relationships Between Sheets

Sheets are arranged in a tree-structured, acyclic, top-down hierarchy. Thus, in general, a sheet has one parent (or no parent) and zero or more children. A sheet may have zero or more siblings (that is, other sheets that share the same parent). In order to describe the relationships between sheets, we need to define some terms.

Adopted A sheet is said to be *adopted* if it has a parent. A sheet becomes the parent of another sheet by adopting that sheet.

Disowned A sheet is said to be *disowned* if it does not have a parent. A sheet ceases to be a child of another sheet by being disowned.

Grafted A sheet is said to be *grafted* when it is part of a sheet hierarchy whose highest ancestor is a graft. In this case, the sheet may be visible on a particular window server.

Degrafted A sheet is said to be *degrafted* when it is part of a sheet hierarchy that cannot possibly be visible on a server, that is, the highest ancestor is not a graft.

Enabled A sheet is said to be *enabled* when it is actively participating in the windowing relationship with its parent. If a sheet is enabled and grafted, and all its ancestors are enabled (they are grafted by definition), then the sheet will be visible if it occupies a portion of the graft region that isn't clipped by its ancestors or ancestor's siblings.

Disabled The opposite of enabled is *disabled*.

7.1.1 Sheet Relationship Functions

The generic functions in this section comprise the sheet protocol. All sheet objects must implement methods for each of these generic functions.

Issue: Is the above statement accurate? — SWM

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

Returns the parent of the *sheet sheet*, or `nil` if the sheet has no parent.

⇒ `sheet-children` *sheet* [*Generic Function*]

Returns a list of sheets that are the children of the *sheet sheet*. Some sheet classes support only a single child; in this case, the result of `sheet-children` will be a list of one element. This function returns objects that reveal CLIM's internal state; do not modify those objects.

⇒ `sheet-adopt-child` *sheet child* [*Generic Function*]

Adds the child sheet *child* to the set of children of the *sheet sheet*, and makes the *sheet* the child's parent. If *child* already has a parent, the `sheet-already-has-parent` error will be signalled.

Some sheet classes support only a single child. For such sheets, attempting to adopt more than a single child will cause the `sheet-supports-only-one-child` error to be signalled.

⇒ `sheet-disown-child` *sheet child* [*Generic Function*]

Removes the child sheet *child* from the set of children of the *sheet sheet*, and makes the parent of the child be `nil`. If *child* is not actually a child of *sheet*, the `sheet-is-not-child` error will be signalled.

⇒ `sheet-siblings` *sheet* [*Generic Function*]

Returns a list of all of the siblings of the *sheet sheet*. The sibling are all of the children of *sheet*'s parent excluding *sheet* itself. This function returns fresh objects that may be modified.

⇒ `sheet-enabled-children` *sheet* [*Generic Function*]

Returns a list of those children of the *sheet sheet* that are enabled. This function returns fresh objects that may be modified.

⇒ `sheet-ancestor-p` *sheet putative-ancestor* [*Generic Function*]

Returns *true* if the the *sheet putative-ancestor* is in fact an ancestor of the *sheet sheet*, otherwise returns *false*.

⇒ `raise-sheet` *sheet* [*Generic Function*]
⇒ `bury-sheet` *sheet* [*Generic Function*]

These functions reorder the children of a sheet by raising the *sheet sheet* to the top or burying it at the bottom. Raising a sheet puts it at the beginning of the ordering; burying it puts it at the end. If sheets overlap, the one that appears “on top” on the display device is earlier in the ordering than the one underneath.

This may change which parts of which sheets are visible on the display device.

⇒ `reorder-sheets` *sheet new-ordering* [*Generic Function*]

Reorders the children of the *sheet sheet* to have the new ordering specified by *new-ordering*. *new-ordering* is an ordered list of the child sheets; elements at the front of *new-ordering* are “on top” of elements at the rear.

If *new-ordering* does not contain all of the children of *sheet*, the `sheet-ordering-underspecified` error will be signalled. If *new-ordering* contains a sheet that is not a child of *sheet*, the `sheet-is-not-child` error will be signalled.

⇒ `sheet-enabled-p` *sheet* [*Generic Function*]

Returns *true* if the the *sheet sheet* is enabled by its parent, otherwise returns *false*. Note that all of a sheet’s ancestors must be enabled before the sheet is viewable.

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

When *enabled-p* is *true*, this enables the the *sheet sheet*. When *enabled-p* is *false*, this disables the sheet.

Note that a sheet is not visible unless it and all of its ancestors are enabled.

⇒ `sheet-viewable-p` *sheet* [*Generic Function*]

Returns *true* if the *sheet sheet* and all its ancestors are enabled, and if one of its ancestors is a graft. See Chapter 9 for further information.

⇒ `sheet-occluding-sheets` *sheet child* [*Generic Function*]

Returns a list of the *sheet child*’s siblings that occlude part or all of the region of the *child*. In general, these are the siblings that are enabled and appear earlier in the *sheet sheet*’s children. If *sheet* does not permit overlapping among its children, `sheet-occluding-sheets` will return `nil`.

This function returns fresh objects that may be modified.

7.1.2 Sheet Genealogy Classes

Different “mixin” classes are provided that implement the relationship protocol.

⇒ `sheet-leaf-mixin` [*Class*]

This class is mixed into sheet classes that will never have children.

⇒ `sheet-single-child-mixin` [*Class*]

This class is mixed into sheet classes that have at most a single child.

⇒ `sheet-multiple-child-mixin` [*Class*]

This class is mixed into sheet classes that may have zero or more children.

7.2 Sheet Geometry

Every sheet has a region and a coordinate system. A sheet’s region refers to its position and extent on the display device, and is represented by some sort of a region object, frequently a rectangle. A sheet’s coordinate system is represented by a coordinate transformation that converts coordinates in its coordinate system to coordinates in its parent’s coordinate system.

7.2.1 Sheet Geometry Functions

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

Returns a transformation that converts coordinates in the *sheet sheet*’s coordinate system into coordinates in its parent’s coordinate system. Using `setf` on this accessor will modify the sheet’s coordinate system, including moving its region in its parent’s coordinate system.

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

Returns a region object that represents the set of points to which the *sheet sheet* refers. The region is in the sheet’s coordinate system. Using `setf` on this accessor modifies the sheet’s region.

Issue: Aren’t there some notification functions that get called when the sheet’s region and transformation are changed? — SWM

Issue: I’m not convinced I like this business of requesting a change by modifying an accessor. It would actually be better to have a `request-` *function, so it would be clear that there might be some delay before the region or transformation was modified. — RSL*

Issue: To reshape and move a region, you generally have to manipulate both of the above. Maybe there should be a single function that takes either or both of a new transformation or region? Maybe region coordinates should be expressed in parents' coordinates, since that's easier to set only one? — RSL

Issue: Using `setf` *on mirrored sheets actually requests that the server move/resize the sheet; the accessor will continue to return the old value until the notification comes in saying that the mirror has been moved. — RSL*

⇒ `map-sheet-point*-to-parent` *sheet x y* [*Generic Function*]

Applies the sheet *sheet*'s transformation to the point (x, y), returning the coordinates of that point in *sheet*'s parent's coordinate system.

⇒ `map-sheet-point*-to-child` *sheet x y* [*Generic Function*]

Applies the inverse of the sheet *sheet*'s transformation to the point (x, y) (represented in *sheet*'s parent's coordinate system), returning the coordinates of that same point in *sheet* coordinate system.

⇒ `map-sheet-bounding-rectangle*-to-parent` *sheet x1 y1 x2 y2* [*Generic Function*]

Applies the sheet *sheet*'s transformation to the bounding rectangle specified by the corner points $(x1, y1)$ and $(x2, y2)$, returning the bounding rectangle of the transformed region as four values, *min-x*, *min-y*, *max-x*, and *max-y*. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for `make-bounding-rectangle`.

⇒ `map-sheet-bounding-rectangle*-to-child` *sheet x1 y1 x2 y2* [*Generic Function*]

Applies the inverse of the sheet *sheet*'s transformation to the bounding rectangle delimited by the corner points $(x1, y1)$ and $(x2, y2)$ (represented in *sheet*'s parent's coordinate system), returning the bounding rectangle of the transformed region as four values, *min-x*, *min-y*, *max-x*, and *max-y*. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for `make-bounding-rectangle`.

⇒ `child-containing-point` *sheet point* [*Generic Function*]
⇒ `child-containing-point*` *sheet x y* [*Generic Function*]

Returns the topmost (earliest) enabled child of the sheet *sheet* whose region contains the point *point* (or the position (x, y)). The point is expressed in *sheet*'s coordinate system.

Issue: What does "earliest" mean exactly in this context? And does it return the earliest child, or the earliest descendant? And does it return an enabled sheet, or any sheet? — SWM

⇒ `children-overlapping-region` *sheet region* [*Generic Function*]
⇒ `children-overlapping-rectangle*` *sheet x1 y1 x2 y2* [*Generic Function*]

Returns the list of enabled children of the sheet *sheet* whose region overlaps the region *region*. `children-overlapping-rectangle*` is a special case of `children-overlapping-region` in which the region is a bounding rectangle whose corner points are $(x1, y1)$ and $(x2, y2)$. The region is expressed in *sheet*'s coordinate system. This function returns fresh objects that may be modified.

⇒ `sheet-delta-transformation` *sheet ancestor* [*Generic Function*]

Returns a transformation that is the composition of all of the sheet transformations between the *sheets sheet* and *ancestor*. If *ancestor* is `nil`, `sheet-delta-transformation` will return the transformation to the root of the sheet hierarchy. If *ancestor* is not an ancestor of sheet, the `sheet-is-not-ancestor` error will be signalled.

The computation of the delta transformation is likely to be cached.

Issue: This used to be called `fetch-delta-transformation`, *right? — SWM*

⇒ `sheet-allocated-region` *sheet child* [*Generic Function*]

Returns the visible region of the *sheet child* in the *sheet sheet*'s coordinate system. If *child* is occluded by any of its siblings, those siblings' regions are subtracted (using `region-difference`) from *child*'s actual region.

7.2.2 Sheet Geometry Classes

Each of the following implements the sheet geometry protocol in a different manner, according to the sheet's requirements.

⇒ `sheet-identity-transformation-mixin` [*Class*]

This class is mixed into sheet classes whose coordinate system is identical to that of its parent.

⇒ `sheet-translation-mixin` [*Class*]

This class is mixed into sheet classes whose coordinate system is related to that of its parent by a simple translation.

⇒ `sheet-y-inverting-transformation-mixin` [*Class*]

This class is mixed into sheet classes whose coordinate system is related to that of its parent by inverting the y coordinate system, and optionally translating by some amount in x and y.

⇒ `sheet-transformation-mixin` [*Class*]

This class is mixed into sheet classes whose coordinate system is related to that of its parent by an arbitrary affine transformation.

7.3 Creating Sheets

Issue: Still needs fleshing out. I'm trying to eliminate contracts from this entire chapter, so I need to figure out how much we want to say here. It may be sufficient to have a function `make-sheet` *that takes a class object and some initialization keywords, and perhaps some interface to the runtime dynamic class stuff. — RSL*

⇒ `make-sheet` *???* [*Function*]

⇒ `make-standard-sheet` *???* [*Function*]

Chapter 8

Sheet Protocols

8.1 Input Protocol

CLIM's windowing substrate provides an input architecture and standard functionality for notifying clients of input that is distributed to their sheets. Input includes such events as the pointer entering and exiting sheets, pointer motion (whose granularity is defined by performance limitations), and pointer button and keyboard events. At this level, input is represented as *event* objects.

Sheets either participate fully in the input protocol or are mute for input. If any functions in the input protocol are called on a sheet that is mute for input, the `sheet-is-mute-for-input` error will be signalled.

Input is processed on a per-port basis by the function `process-next-event`. In multiprocessing environments, a process that calls `process-next-event` in a loop is created for each port. In single-process Lisps, `process-next-event` is called whenever the user would go blocked for input.

`process-next-event` has three main tasks when it receives an event. First, it must determine to which *client* the event is addressed. Typically, the client is a sheet, but there are other special-purpose clients to which events can also be dispatched. Next, `process-next-event` formats the event into a standard format, and finally it *distributes* the event to the client. A client may then either handle the event synchronously, or it may queue it for later handling by another process.

Input events can be broadly categorized into *pointer events* and *keyboard events*. By default, pointer events are dispatched to the lowest sheet in the hierarchy whose region contains the location of the pointer. Keyboard events are dispatched to the port's keyboard input focus; the accessor `port-keyboard-input-focus` contains the event client that receives the port's keyboard events.

Issue: Can someone explain to me the exact distinction between dispatching, distributing, and processing? — SWM

8.1.1 Input Protocol Functions

Issue: Do we want to advertise `sheet-event-queue`*? Probably not. — SWM*

⇒ `process-next-event` *port* `&key` *wait-function timeout* [*Generic Function*]

This function provides a standard interface for one pass through a port's event processing loop. *wait-function* is either `nil` or a function of no arguments that acts as a predicate; it has dynamic extent. The predicate should wait until one of three conditions occurs:

- If an event if received and processed, the predicate should return *true*.
- If a timeout occurs, the predicate should return *false*.
- If the wait function returns *true*, the predicate should return the two values *false* and `:timeout`.

A port implementation must provide a method for this function that reads the next window server-specific device event, blocking if necessary, and then invokes the event distributor.

⇒ `port-keyboard-input-focus` *port* [*Generic Function*]
⇒ `(setf port-keyboard-input-focus)` *focus port* [*Generic Function*]

Returns the client to which keyboard events are to be dispatched.

Issue: Name of this accessor? Maybe it should just be `keyboard-input-focus`*. We might like to have this object be a frame, which in turn might also want to have a per-frame focus sheet or pane or mosaic or whatever. — RSL*

⇒ `distribute-event` *port event* [*Generic Function*]

The *event* is distributed to the *port*'s proper client. In general, this will be the keyboard input focus for keyboard events, and the lowest sheet under the pointer for pointer events.

Issue: Do we just want to call this function `dispatch-event` *and have it called on the port first? — RSL*

⇒ `dispatch-event` *client event* [*Generic Function*]

This function is called by `process-next-event` to inform a client about an event of interest. It is invoked synchronously by whatever process called `process-next-event`, so many methods for this function will simply queue the event for later handling. Certain classes of clients and events may cause this function immediately to call either `queue-event` or `handle-event`, or to ignore the event entirely.

Issue: What exactly is a "client"? It says above that it can be something other than a sheet, but what are those other things? — SWM

⇒ `queue-event` *client event* [*Generic Function*]

Places the event *event* into the queue of events for the client *client*.

⇒ `handle-event` *client event* [*Generic Function*]

Implements the client's policy with respect to the event. For example, if the programmer wishes to highlight a sheet in response to an event that informs it that the pointer has entered its territory, there would be a method to carry out the policy that specializes the appropriate sheet and event classes.

In addition to `queue-event`, the queued input protocol handles the following generic functions:

Issue: These names are subject to change. — RSL

⇒ `event-read` *client* [*Generic Function*]

Takes the next event out of the queue of events for this client.

⇒ `event-read-no-hang` *client* [*Generic Function*]

Takes the next event out of the queue of events for this client. It returns `nil` if there are no events in the queue.

⇒ `event-peek` *client* `&optional` *event-type* [*Generic Function*]

Returns the next event in the queue without removing it from the queue. If *event-type* is supplied, events that are not of that type are first removed and discarded.

⇒ `event-unread` *client event* [*Generic Function*]

Places the *event* at the head of the *client*'s event queue, so as to be the next event read.

⇒ `event-listen` *client* [*Generic Function*]

Returns *true* if there are any events queued for *client*, otherwise returns *false*.

8.1.2 Input Protocol Classes

Issue: Maybe these names should be `-client-` *and not* `-sheet-`*? For example,* `standard-input-client` *instead of* `standard-sheet-input-mixin`*? — RSL*

Most classes of sheets will have one of the following input protocol classes mixed in. Of course, a sheet can always have a specialized method for a specific class of event that will override the default. For example, a sheet may need to have only pointer click events dispatched to itself, and may delegate all other events to some other input client. Such a sheet should have `delegate-sheet-input-mixin` as a superclass, and have a more specific method for `dispatch-event` on its class and `click-event`.

⇒ `standard-sheet-input-mixin` [*Class*]

This class of sheet provides a method for `dispatch-event` that calls `queue-event` on each device event. Note that configuration events invoke `handle-event` immediately.

⇒ `immediate-sheet-input-mixin` [*Class*]

Issue: Is this a good name? RWK proposed this. Anybody have other opinions? — RSL

This class of sheet provides a method for `dispatch-event` that calls `handle-event` immediately for all events.

⇒ `mute-sheet-input-mixin` [*Class*]

⇒ `delegate-sheet-input-mixin` [*Class*]

Issue: Is this a good name? — RSL

This class of sheet provides a method for `dispatch-event` that calls `dispatch-event` on a designated substitute and the event. The initialization argument `:delegate` or the accessor delegate-sheet-delegate may be used to set the recipient of dispatched events.

Issue: What is a "substitute" and how do you "designate" it? Is the substitute what `delegate-sheet-delegate` *returns? Does't this want an initarg for the substitute? — SWM*

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

This may be set to another recipient of events dispatched to a sheet of class `delegate-sheet-input-mixin`. If the delegate is `nil`, events are discarded.

8.2 Standard Device Events

An *event* is a CLIM object that represents some sort of user gesture (such as moving the pointer or pressing a key on the keyboard) or that corresponds to some sort of notification from the display server. Event objects store such things as the sheet associated with the event, the x and y bposition of the pointer within that sheet, the key name or character corresponding to a key on the keyboard, and so forth.

Figure 8.1 shows all the event classes.

⇒ `event` [*Protocol Class*]

The protocol class that corresponds to any sort of "event". If you want to create a new class that behaves like an event, it should be a subclass of `event`. Subclasses of `event` must obey the event protocol.

All of the event classes are immutable.

⇒ `eventp` *object* [*Function*]

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

⇒ `:timestamp` [*Init Arg*]

All subclasses of `event` must take a `:timestamp` initarg, which is used to specify the timestamp for the event.

```
event
  device-event
    keyboard-event
      key-press-event
      key-release-event
    pointer-event
      pointer-button-event
        pointer-button-press-event
        pointer-button-release-event
        pointer-button-hold-event
      pointer-motion-event
        pointer-enter-event
        pointer-exit-event
  window-event
    window-configuration-event
    window-repaint-event
  timer-event
```

Figure 8.1: CLIM event classes. All classes that appear at a given indentation are subclasses of the class that appears above and at a lesser indentation.

⇒ `event-timestamp` *event* [*Generic Function*]

Returns an integer that is a monotonically increasing timestamp for the the *event event*. The timestamp must have at least as many bits of precision as a fixnum.

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

For the *event event*, returns a keyword with the same name as the class name, except stripped of the "-event" ending. For example, the keyword `:key-press` is returned by `event-type` for an event whose class is `key-press-event`.

All event classes must implement methods for `event-type` and `event-timestamp`.

⇒ `device-event` [*Class*]
⇒ `:sheet` [*Init Arg*]

⇒ `:modifier-key-state` [*Init Arg*]

The class that corresponds to any sort of device event. This is a subclass of `event`.

All subclasses of `device-event` must take the `:sheet` and `:modifier-key-state` initargs, which are used to specify the sheet and modifier key state components for the event.

⇒ `event-sheet` *device-event* [*Generic Function*]

Returns the sheet associated with the event *device-event*.

⇒ `event-modifier-key-state` *device-event* [*Generic Function*]

Returns a value that encodes the state of all the modifier keys on the keyboard. This will be a mask consisting of the `logior` of `+shift-key+`, `+control-key+`, `+meta-key+`, `+super-key+`, and `+hyper-key+`.

All device event classes must implement methods for `event-sheet` and `event-modifier-key-state`.

⇒ `keyboard-event` [*Class*]
⇒ `:key-name` [*Init Arg*]

The class that corresponds to any sort of keyboard event. This is a subclass of `device-event`.

All subclasses of `keyboard-event` must take the `:key-name` initarg, which is used to specify the key name component for the event.

⇒ `keyboard-event-key-name` *keyboard-event* [*Generic Function*]

Returns the name of the key that was pressed or released in a keyboard event. This will be a symbol whose value is port-specific. Key names corresponding to the set of "standard" characters (such as the alphanumerics) will be a symbol in the keyword package.

⇒ `keyboard-event-character` *keyboard-event* [*Generic Function*]

Returns the character associated with the event *keyboard-event*, if there is any.

All keyboard event classes must implement methods for `keyboard-event-key-name` and `keyboard-event-character`.

⇒ `key-press-event` [*Class*]
⇒ `key-release-event` [*Class*]

The classes that correspond to a key press or release event. This is a subclass of `keyboard-event`.

⇒ `pointer-event` [*Class*]
⇒ `:pointer` [*Init Arg*]

⇒ `:button` [*Init Arg*]

⇒ `:x` [*Init Arg*]

⇒ `:y` [*Init Arg*]

The class that corresponds to any sort of pointer event. This is a subclass of `device-event`.

All subclasses of `pointer-event` must take the `:pointer`, `:button`, `:x`, and `:y` initargs, which are used to specify the pointer object, pointer button, and native x and y position of the pointer at the time of the event. The sheet's x and y positions are derived from the supplied native x and y positions and the sheet itself.

⇒ `pointer-event-x` *pointer-event* [*Generic Function*]
⇒ `pointer-event-y` *pointer-event* [*Generic Function*]

Returns the x and y position of the pointer at the time the event occurred, in the coordinate system of the sheet that received the event. All pointer events must implement a method for these generic functions.

⇒ `pointer-event-native-x` *pointer-event* [*Generic Function*]
⇒ `pointer-event-native-y` *pointer-event* [*Generic Function*]

Returns the x and y position of the pointer at the time the event occurred, in the pointer's native coordinate system. All pointer events must implement a method for these generic functions.

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

Returns the pointer object to which this event refers.

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

Returns the number of the pointer button that was pressed, which will be one of `+pointer-left-button+`, `+pointer-middle-button+`, or `+pointer-right-button+`.

All pointer event classes must implement methods for `pointer-event-x`, `pointer-event-y`, `pointer-event-native-x`, `pointer-event-native-y`, `pointer-event-pointer`, and `pointer-event-button`.

⇒ `pointer-button-event` [*Class*]

The class that corresponds to any sort of pointer button event. This is a subclass of `pointer-event`.

⇒ `pointer-button-press-event` [*Class*]
⇒ `pointer-button-release-event` [*Class*]
⇒ `pointer-button-hold-event` [*Class*]

The classes that correspond to a pointer button press, button release, and click-and-hold events. These are subclasses of `pointer-button-event`.

⇒ `pointer-motion-event` [*Class*]

The class that corresponds to any sort of pointer motion event. This is a subclass of `pointer-event`.

⇒ `pointer-enter-event` [*Class*]
⇒ `pointer-exit-event` [*Class*]

The classes that correspond to a pointer enter or exit event. This is a subclass of `pointer-motion-event`.

⇒ `window-event` [*Class*]
⇒ `:region` [*Init Arg*]

The class that corresponds to any sort of windowing event. This is a subclass of `device-event`.

All subclasses of `window-event` must take a `:region` initarg, which is used to specify the damage region associated with the event.

⇒ `window-event-region` *window-event* [*Generic Function*]

Returns the region of the sheet that is affected by a window event.

⇒ `window-event-native-region` *window-event* [*Generic Function*]

Returns the region of the sheet in native coordinates.

⇒ `window-event-mirrored-sheet` *window-event* [*Generic Function*]

Returns the mirrored sheet that is attached to the mirror on which the event occurred.

All window event classes must implement methods for `window-event-region`, `window-event-native-region`, and `window-event-mirrored-sheet`.

⇒ `window-configuration-event` [*Class*]

The class that corresponds to a window changing its size or position. This is a subclass of `window-event`.

⇒ `window-repaint-event` [*Class*]

The class that corresponds to a request to repaint the window. This is a subclass of `window-event`.

⇒ `timer-event` [*Class*]

The class that corresponds to a timeout event. This is a subclass of `event`.

⇒ `+pointer-left-button+` [*Constant*]
⇒ `+pointer-middle-button+` [*Constant*]
⇒ `+pointer-right-button+` [*Constant*]

Constants that correspond to the left, middle, and right button on a pointing device. `pointer-event-button` will returns one of these three values.

⇒ `+shift-key+` [*Constant*]
⇒ `+control-key+` [*Constant*]
⇒ `+meta-key+` [*Constant*]
⇒ `+super-key+` [*Constant*]
⇒ `+hyper-key+` [*Constant*]

Constants that correspond to the shift, control, meta, super, and hyper modifier keys being held down on the keyboard. These constants must be powers of 2 so that they can be combined with `logior` and tested with `logtest`. `event-modifier-key-state` will return some combination of these values.

Implementations must support at least shift, control, and meta modifiers. Control and meta might correspond to the control and option or command shift keys on a Macintosh keyboard, for example.

⇒ `key-modifier-state-match-p` *button modifier-state* `&body` *clauses* [*Macro*]

This macro generates code that will check whether the modifier state *modifier-state* and the pointer button *button* match all of the clauses. *clauses* are implicitly grouped by `and`. Matching a button or a modifier means that that the modifier state indicates that the button or modifier is pressed.

A clause may be one of:

- A pointer button. One of `:left`, `:middle` or `:right`.
- A modifier key. One of `:shift`, `:control`, `:meta`, `:super` or `:hyper`.
- `(and [`*clause*`]+)`
- `(or [`*clause*`]+)`
- `(not` *clause*`)`

8.3 Output Protocol

The output protocol is concerned with the appearance of displayed output on the window associated with a sheet.

Sheets either participate fully in the output protocol or are mute for output. If any functions in the output protocol are called on a sheet that is mute for output, the `sheet-is-mute-for-output` error will be signalled.

8.3.1 Output Properties

Each sheet retains some output state that logically describes how output is to be rendered on its window. Such information as the foreground and background ink, line thickness, and transformation to be used during drawing are provided by this state. This state may be stored in a *medium* associated with the sheet itself, be derived from a parent, or may have some global default, depending on the sheet itself.

If a sheet is mute for output, it is an error to set any of these values.

⇒ `medium` [*Protocol Class*]

The protocol class that corresponds to the output state for some kind of sheet. There is no single advertised standard medium class. If you want to create a new class that behaves like a medium, it should be a subclass of `medium`. Subclasses of `medium` must obey the medium protocol.

⇒ `mediump` *object* [*Function*]

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

The following generic functions comprise the basic medium protocol. All mediums must implement methods for these generic functions. Often, a sheet class that supports the output protocol will implement a "trampoline" method that passes the operation on to `sheet-medium` of the sheet.

⇒ `medium-foreground` *medium* [*Generic Function*]
⇒ `(setf medium-foreground)` *design medium* [*Generic Function*]

Returns (and, with `setf`, sets) the current foreground ink for the *medium medium*. This is described in detail in Chapter 10.

⇒ `medium-background` *medium* [*Generic Function*]
⇒ `(setf medium-background)` *design medium* [*Generic Function*]

Returns (and, with `setf`, sets) the current background ink for the *medium medium*. This is described in detail in Chapter 10.

⇒ `medium-ink` *medium* [*Generic Function*]
⇒ `(setf medium-ink)` *design medium* [*Generic Function*]

Returns (and, with `setf`, sets) the current drawing ink for the *medium medium*. This is described in detail in Chapter 10.

⇒ `medium-transformation` *medium* [*Generic Function*]
⇒ `(setf medium-transformation)` *transformation medium* [*Generic Function*]

Returns (and, with `setf`, sets) the "user" transformation that converts the coordinates presented to the drawing functions by the programmer to the *medium medium*'s coordinate system. By default, it is the identity transformation. This is described in detail in Chapter 10.

⇒ `medium-clipping-region` *medium* [*Generic Function*]
⇒ `(setf medium-clipping-region)` *region medium* [*Generic Function*]

Returns (and, with `setf`, sets) the clipping region that encloses all output performed on the *medium medium*. It is returned and set in user coordinates. That is, to convert the user clipping region to medium coordinates, it must be transformed by the value of `medium-transformation`. For example, the values returned by

```
(let (cr1 cr2)
  ;; Ensure that the sheet's clipping region and transformation will be reset:
  (with-drawing-options (sheet :transformation +identity-transformation+
                               :clipping-region +everywhere+)
    (setf (medium-clipping-region sheet) (make-rectangle* 0 0 10 10))
    (setf (medium-transformation sheet) (clim:make-scaling-transformation 2 2))
    (setf cr1 (medium-clipping-region sheet))
    (setf (medium-clipping-region sheet) (make-rectangle* 0 0 10 10))
    (setf (medium-transformation sheet) +identity-transformation+)
    (setf cr2 (medium-clipping-region sheet))
    (values cr1 cr2)))
```

are two rectangles. The first one has edges of (0,0,5,5), while the second one has edges of (0,0,20,20).

By default, the user clipping region is the value of `+everywhere+`.

Issue: What are "user coordinates"? We need to define all of the coordinate systems in one place: device, window, stream, etc. — SWM

⇒ `medium-line-style` *medium* [*Generic Function*]
⇒ `(setf medium-line-style)` *line-style medium* [*Generic Function*]

Returns (and, with `setf`, sets) the current line style for the *medium medium*. This is described in detail in Chapter 10 and Section 10.3.

⇒ `medium-text-style` *medium* [*Generic Function*]
⇒ `(setf medium-text-style)` *text-style medium* [*Generic Function*]

Returns (and, with `setf`, sets) the current text style for the *medium medium* of any textual output that may be displayed on the window. This is described in detail in Chapter 10.

⇒ `medium-default-text-style` *medium* [*Generic Function*]
⇒ `(setf medium-default-text-style)` *text-style medium* [*Generic Function*]

Returns (and, with `setf`, sets) the default text style for output on the *medium medium*. This is described in detail in Chapter 10.

⇒ `medium-merged-text-style` *medium* [*Generic Function*]

Returns the actual text style used in rendering text on the *medium medium*. It returns the result of

```
(merge-text-styles (medium-text-style medium)
                   (medium-default-text-style medium))
```

Thus, those components of the current text style that are not `nil` will replace the defaults from medium's default text style. Unlike the preceding text style function, `medium-merged-text-style` is read-only.

8.3.2 Output Protocol Functions

The output protocol functions on mediums (and sheets that support the standard output protocol) include those functions described in Section 12.6.

Issue: Need to do better here — SWM

8.3.3 Output Protocol Classes

Issue: To be written. — RSL

⇒ `standard-sheet-output-mixin` [*Class*]

⇒ `primitive-sheet-output-mixin` [*Class*]

(No client transformation/clipping region, no foreground/background inks, no text styles.)

⇒ `mute-sheet-output-mixin` [*Class*]

⇒ `permanent-medium-sheet-output-mixin` [*Class*]

⇒ `temporary-medium-sheet-output-mixin` [*Class*]

8.3.4 Associating a Medium with a Sheet

Before a sheet may be used for output, it must be associated with a medium. Some sheets are permanently associated with media for output efficiency; for example, CLIM window stream sheets have a medium that is permanently allocated to the window.

However, many kinds of sheets only perform output infrequently, and therefore do not need to be associated with a medium except when output is actually required. Sheets without a permanently associated medium can be much more lightweight than they otherwise would be. For example, in a program that creates a sheet for the purpose of displaying a border for another sheet, the border sheet receives output only when the window's shape is changed.

To associate a sheet with a medium, the macro `with-sheet-medium` is used.

⇒ `with-sheet-medium` *(medium sheet)* `&body` *body* [*Macro*]

Within the body, the variable *medium* is bound to the sheet's medium. If the sheet does not have a medium permanently allocated, one will be allocated, associated with the sheet for the duration of the body, and deallocated as the when the body has been exited. The values of the last form of the body are returned as the values of `with-sheet-medium`.

The *medium* argument is not evaluated, and must be a symbol that is bound to a medium. *body* may have zero or more declarations as its first forms.

⇒ `with-sheet-medium-bound` *(sheet medium)* `&body` *body* [*Macro*]

`with-sheet-medium-bound` is used to associate the specific medium *medium* with the sheet *sheet* for the duration of the body *body*. Typically, a single medium will be allocated an passed to several different sheets that can use the same medium.

If the sheet already has a medium allocated to it, the new medium will not be given to the sheet. If the value of *medium* is `nil`, `with-sheet-medium-bound` is exactly equivalent to `with-sheet-medium`. The values of the last form of the body are returned as the values of `with-sheet-medium-bound`.

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

⇒ `sheet-medium` *sheet* [*Generic Function*]

Returns the medium associated with the *sheet sheet*. If *sheet* does not have a medium allocated to it, `sheet-medium` returns `nil`.

8.4 Repaint Protocol

The repaint protocol is the mechanism whereby a program keeps the display up-to-date, reflecting the results of both synchronous and asynchronous events. The repaint mechanism may be invoked by user programs each time through their top-level command loop. It may also be invoked directly or indirectly as a result of events received from the display server host. For example, if a window is on display with another window overlapping it, and the second window is buried, a "damage notification" event may be sent by the server; CLIM would cause a repaint to be executed for the newly-exposed region.

8.4.1 Repaint Protocol Functions

⇒ `dispatch-repaint` *sheet region* [*Generic Function*]

Requests a repaint of the *region region* within the *sheet sheet*. The region is given in sheet coordinates. Depending on the class of the sheet, this may cause `queue-repaint` or `handle-repaint` to be invoked.

All CLIM implementations must support repainting for regions that are rectangles or region sets composed entirely of rectangles.

⇒ `queue-repaint` *sheet region* [*Generic Function*]

Requests that a repaint event for the *region region* be placed in the input queue of the *sheet sheet*. A program that reads events out of the queue will be expected to call `handle-event` for the repaint region; the method for that generic function on repaint events will generally call `repaint-sheet`.

⇒ `handle-repaint` *sheet medium region* [*Generic Function*]

Recursively causes repainting of the *sheet sheet* and any of its children that overlap the *region region*. *medium* is the medium to use for the repainting; if it is `nil`, `handle-repaint` will allocate a medium and associate it with the sheet. `handle-repaint` will call `repaint-sheet` on *sheet*, and then call `handle-repaint` on all of the children of *sheet*.

⇒ `repaint-sheet` *sheet region* [*Generic Function*]

Implements repainting for a given sheet class. It may only be called on a sheet that has an associated medium. *sheet* and *region* are as for `dispatch-repaint`.

8.4.2 Repaint Protocol Classes

⇒ `standard-repainting-mixin` [*Class*]

Defines a `dispatch-repaint` method that calls `queue-repaint`.

⇒ `immediate-repainting-mixin` [*Class*]

Issue: Is this a good name? — RSL

Defines a `dispatch-repaint` method that calls `handle-repaint`.

⇒ `mute-repainting-mixin` [*Class*]

Defines a `dispatch-repaint` method that calls `queue-repaint`, and a method on `repaint-sheet` that does nothing. This means that its children will be recursively repainted when the repaint event is handled.

8.5 Sheet Notification Protocol

The notification protocol allows sheet clients to be notified when a sheet hierarchy is changed. Sheet clients can observe modification events by providing `:after` methods for functions defined by this protocol.

8.5.1 Relationship to Window System Change Notifications

Issue: To be written. — RSL

⇒ `note-sheet-grafted` *sheet* [*Generic Function*]
⇒ `note-sheet-degrafted` *sheet* [*Generic Function*]
⇒ `note-sheet-adopted` *sheet* [*Generic Function*]
⇒ `note-sheet-disowned` *sheet* [*Generic Function*]
⇒ `note-sheet-enabled` *sheet* [*Generic Function*]
⇒ `note-sheet-disabled` *sheet* [*Generic Function*]

These notification functions are invoked when the state change has been made to the *sheet sheet*.

8.5.2 Geometry Notifications

Issue: To be written. — RSL

⇒ `note-sheet-region-changed` *sheet* [*Generic Function*]
⇒ `note-sheet-transformation-changed` *sheet* [*Generic Function*]

These notification functions are invoked when the region or transformation of the *sheetsheet* has been changed. When the regions and transformations of a sheet are changed directly, the client is required to call `note-sheet-region-changed` or `note-sheet-transformation-changed`.

Chapter 9

Ports, Grafts, and Mirrored Sheets

9.1 Introduction

A sheet hierarchy must be attached to a display server so as to permit input and output. This is managed by the use of ports and grafts.

9.2 Ports

A *port* is a logical connection to a display server. It is responsible for managing display output and server resources, and for handling incoming input events. Typically, the programmer will create a single port that will manage all of the windows on the display.

A port is described with a *server path*. A server path is a list whose first element is a keyword that selects the kind of port. The remainder of the server path is a list of alternating keywords and values whose interpretation is port type-specific.

⇒ `find-port &key` *(server-path* `*default-server-path*`*)* [*Function*]

Finds a port that provides a connection to the window server addressed by *server-path*. If no such connection exists, a new connection will be constructed and returned.

The following server paths are currently supported:

⇒ `:x11 &key` *host display-id screen-id* [*Server Path*]

Given this server path, `find-port` finds a port for the X server on the given *host*, using the *display-id* and *screen-id*.

On a Unix host, if these values are not supplied, the defaults come from the `display` environment variable. Each CLIM implementation must describe how it uses such environment variables.

⇒ `:genera &key` *screen* [*Server Path*]

Issue: This is old and obsolete. Ditto `:coral`*, or whatever we're going to call it now. — RSL*

⇒ `*default-server-path*` [*Variable*]

This special variable is used by `find-port` and its callers to default the choice of a display service to locate. Binding this variable in a dynamic context will affect the defaulting of this argument to these functions. This variable will be defaulted according to the environment. In the Unix environment, for example, CLIM will attempt to set this variable based on the value of the `display` environment variable.

⇒ `sheet-port` *sheet* [*Generic Function*]

Returns the port associated with the sheet *sheet*. `sheet-port` is defined for all kinds of sheets (including grafts and streams that support the CLIM graphics protocol) and mediums. For degrafted sheets or other objects that aren't currently associated with particular ports, `sheet-port` will return `nil`.

⇒ `with-port-locked` *port* `&body` *body* [*Macro*]

Executes *body* after grabbing a lock associated with the port *port*, which may be a port or any object on which the function `sheet-port` works. If *object* currently has no port, *body* will be executed without locking.

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

⇒ `port-properties` *port indicator* [*Generic Function*]
⇒ `(setf port-properties)` *property port indicator* [*Generic Function*]

These functions provide a port-based property list. They are primarily intended to support users of CLIM that may need to associate certain information with ports. For example, the implementor of a special graphics package may need to maintain resource tables for each port on which it is used.

⇒ `map-over-ports` *function* [*Function*]

Invokes *function* on each existing port. Function is a function of one argument, the port; it has dynamic extent.

⇒ `restart-port` *port* [*Generic Function*]

In a multi-process Lisp, `restart-port` restarts the global input processing loop associated with the port *port*. All pending input events are discarded. Server resources may or may not be released and reallocated during or after this action.

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

Destroys the connection with the window server represented by the port *port*. All sheet hierarchies that are associated with *port* are forcibly degrafted by disowning the children of grafts on *port* using *sheet-disown-child*. All server resources utilized by such hierarchies or by any graphics objects on *port* are released as part of the connection shutdown.

⇒ `add-watcher` *port watcher* [*Generic Function*]
⇒ `delete-watcher` *port watcher* [*Generic Function*]
⇒ `reset-watcher` *watcher how* [*Generic Function*]

Watchers are a mechanism for clients to be notified when the port is destroyed or restarted. `add-watcher` and `delete-watcher` provide an interface for adding and removing *watcher* to the list of watchers that the *port port* maintains. `destroy-port` and `restart-port` invoke `reset-watcher` on each of these watchers with `:destroy` and `:restart` being passed in as *how*, respectively.

Issue: What exactly is a watcher? The above is not specific enough. — SWM

9.3 Grafts

A graft is a special sheet that is directly connected to a display server. Typically, a graft is the CLIM sheet that represents the root window of the display. There may be several grafts that are all attached to the same root window; these grafts may have differing coordinate systems.

To display a sheet on a display, it must have a graft for an ancestor. In addition, the sheet and all of its ancestors must be enabled, including the graft. In general, a sheet becomes grafted when it (or one of its ancestors) is adopted by a graft.

⇒ `sheet-grafted-p` *sheet* [*Generic Function*]

Returns *true* if any of the sheet's ancestors is a graft, otherwise returns *false*.

⇒ `find-graft &key` *(port* `(find-port)`*) (server-path* `*default-server-path*`*) (orientation* `:default`*) (units* `:device`*)* [*Function*]

Finds a graft that represents the display device on the port *port* that also matches the other supplied parameters. If no such graft exists, a new graft is constructed and returned.

If *server-path* is supplied, `find-graft` finds a graft whose port provides a connection to the window server addressed by *server-path*.

It is an error to provide both *port* and *server-path* in a call to `find-graft`.

orientation specifies the orientation of the graft's coordinate system. Supported values are `:default` and `:graphics`, which have the meanings describe below:

- `:default`—a coordinate system with its origin is in the upper left hand corner of the display device with y increasing from top to bottom and x increasing from left to right.
- `:graphics`—a coordinate system with its origin in the lower left hand corner of the display device with y increasing from bottom to top and x increasing from left to right.

units specifies the units of the coordinate system and defaults to `:device`, which means the device units of the host window system (such as pixels). Other supported values include `:inches`, `:millimeters`, and `:screen-sized`, which means that one unit in each direction is the width and height of the display device.

Issue: I don't know how much of this is obsolete. — RSL

⇒ `sheet-graft` *sheet* [*Generic Function*]

Returns the graft currently associated with the sheet *sheet*. `sheet-graft` is defined for all kinds of sheets (including streams that support the CLIM graphics protocol) and mediums. For degrafted sheets or other objects that aren't currently associated with a particular graft, `sheet-graft` will return `nil`.

⇒ `map-over-grafts` *function port* [*Function*]

Invokes *function* on each existing graft associated with the port *port*. *function* is a function of one argument, the graft; it has dynamic extent.

⇒ `with-graft-locked` *graft* `&body` *body* [*Macro*]

Executes *body* after grabbing a lock associated with the *graft graft*, which may be a graft or any object on which the function `sheet-graft` works. If *object* currently has no graft, *body* will be executed without locking.

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

⇒ `graft-orientation` *graft* [*Generic Function*]

Returns the orientation of the *graft graft*'s coordinate system. The returned value will be either `:default` or `:graphics`. The meanings of these values are the same as described for the orientation argument to `find-graft`.

⇒ `graft-units` *graft* [*Generic Function*]

Returns the units of the *graft graft*'s coordinate system. The returned value will be one of `:device`, `:inches`, `:millimeters`, or `:screen-sized`. The meanings of these values are the same as described for the units argument to `find-graft`.

⇒ `graft-width` *graft* `&key` *(units* `:device`*)* [*Generic Function*]
⇒ `graft-height` *graft* `&key` *(units* `:device`*)* [*Generic Function*]

Returns the width and height of the *graft graft* (and by extension the associated host window) in the units indicated. *Units* may be any of `:device`, `:inches`, `:millimeters`, or `:screen-sized`. The meanings of these values are the same as described for the units argument to `find-graft`. Note if a *unit* of `:screen-sized` is specified, both of these functions will return a value of `1`.

⇒ `graft-pixels-per-millimeter` *graft* [*Function*]
⇒ `graft-pixels-per-inch` *graft* [*Function*]

Returns the number of pixels per millimeter or inch of the *graft graft*. These functions are provided as a convenience to programmers and can be easily written in terms of `graft-width` or `graft-height`.

Issue: Do we want to support non-square pixels? If so, these functions aren't sufficient. — Rao

9.4 Mirrors and Mirrored Sheets

A *mirrored sheet* is a special class of sheet that is attached directly to a window on a display server. Grafts, for example, are always mirrored sheets. However, any sheet anywhere in a sheet hierarchy may be a mirrored sheet. A mirrored sheet will usually contain a reference to a window system object, called a mirror. For example, a mirrored sheet attached to an X11 server might have an X window system object stored in one of its slots. Allowing mirrored sheets at any point in the hierarchy enables the adaptive toolkit facilities.

Since not all sheets in the hierarchy have mirrors, there is no direct correspondence between the sheet hierarchy and the mirror hierarchy. However, on those display servers that support hierarchical windows, the hierarchies must be parallel. If a mirrored sheet is an ancestor of another mirrored sheet, their corresponding mirrors must have a similar ancestor/descendant relationship.

CLIM interacts with mirrors when it must display output or process events. On output, the mirrored sheet closest in ancestry to the sheet on which we wish to draw provides the mirror on which to draw. The mirror's drawing clipping region is set up to be the intersection of the user's clipping region and the sheet's region (both transformed to the appropriate coordinate system) for the duration of the output. On input, events are delivered from mirrors to the sheet hierarchy. The CLIM port must determine which sheet shall receive events based on information such as the location of the pointer.

In both of these cases, we must have a coordinate transformation that converts coordinates in the mirror (so-called "native" coordinates) into coordinates in the sheet and vice-versa.

9.4.1 Mirror Functions

⇒ `sheet-direct-mirror` *sheet* [*Generic Function*]

Returns the mirror of the *sheet sheet*. If the sheet is not mirrored (or does not currently have a mirror), `sheet-mirror` returns `nil`.

⇒ `sheet-mirrored-ancestor` *sheet* [*Generic Function*]

Returns the nearest mirrored ancestor of the *sheet sheet*.

⇒ `sheet-mirror` *sheet* [*Generic Function*]

Returns the mirror of the *sheet sheet*. If the sheet is not itself mirrored, `sheet-mirror` returns the direct mirror of its nearest mirrored ancestor. `sheet-mirror` could be implemented as:

```
(defun sheet-mirror (sheet)
  (sheet-direct-mirror (sheet-mirrored-ancestor sheet)))
```

⇒ `realize-mirror` *port mirrored-sheet* [*Generic Function*]

Creates a mirror for the sheet *mirrored-sheet* on the port *port*, if it does not already have one. The returned value is the sheet's mirror.

9.4.2 Internal Interfaces for Native Coordinates

⇒ `sheet-native-transformation` *sheet* [*Generic Function*]

Returns the transformation for the sheet *sheet* that converts sheet coordinates into native coordinates. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

⇒ `sheet-native-region` *sheet* [*Generic Function*]

Returns the region for the sheet *sheet* in native coordinates. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

Issue: These used to be called `fetch-mirrored-sheet`, `fetch-native-transformation`, *and* `fetch-native-clipping-region`. *— RSL*

⇒ `sheet-device-transformation` *sheet* [*Generic Function*]

Returns the transformation used by the graphics output routines when drawing on the mirror. This is the composition of the sheet's native transformation and the user transformation. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

⇒ `sheet-device-region` *sheet* [*Generic Function*]

Returns the actual clipping region to be used when drawing on the mirror. This is the intersection of the user's clipping region (transformed by the device transformation) with the sheet's native region. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

⇒ `invalidate-cached-transformations` *sheet* [*Generic Function*]

`sheet-native-transformation` and `sheet-device-transformation` typically cache the transformations for performance reasons. `invalidate-cached-transformations` clears the cached native and device values for the sheet *sheet*'s transformation and clipping region. It is invoked when a sheet's native transformation changes, which happens when a sheet's transformation is changed or when `invalidate-cached-transformations` is called on any of its ancestors.

⇒ `invalidate-cached-regions` *sheet* [*Generic Function*]

`sheet-native-region` and `sheet-device-region` typically cache the regions for performance reasons. `invalidate-cached-regions` clears the cached native and device values for the sheet *sheet*'s native clipping region. It is invoked when a sheet's native clipping region changes, which happens when the clipping region changes or when `invalidate-cached-regions` is called on any of its ancestors.

Part IV

Sheet and Medium Output Facilities

Chapter 10

Drawing Options

This chapter describes the drawing options that are used by CLIM's drawing functions, and the relationship between drawing options, sheets, and mediums. These drawing options control various aspects of the drawing process, and can be provided as keyword arguments to all of the drawing functions.

10.1 Medium Components

Medium objects contain components that correspond to the drawing options; when no value for a drawing option is explicitly provided to a drawing function, it is taken from the medium. These values can be directly queried or modified using accessors defined on the sheet or medium. They can also be temporarily bound within a dynamic context using `with-drawing-options`, `with-text-style`, and related forms.

`setf` of one of these components while it is temporarily bound (via `with-drawing-options`, for instance) takes effect immediately but is undone when the dynamic binding context is exited.

In systems that support multiple processes, the consequences are unspecified if one process reads or writes a medium component that is temporarily bound by another process.

The following functions read and write components of a medium related to drawing options. While these functions are defined for mediums, they can also be called on sheets support the sheet output protocol and on streams that output to such sheets. All classes that support the medium protocol must implement methods for these generic functions. Often, a sheet class that supports the output protocol will implement a "trampoline" method that passes the operation on to `sheet-medium` of the sheet.

⇒ `medium-foreground` *medium* [*Generic Function*]
⇒ `medium-background` *medium* [*Generic Function*]

Returns the foreground and background inks (which are designs) for the *medium medium*, respectively. The foreground ink is the default ink used when drawing. The background ink is the ink used when erasing. See Chapter 13 for a more complete description of designs.

Any indirect inks are resolved against the foreground and background at the time a design is rendered.

⇒ `(setf medium-foreground)` *design medium* [*Generic Function*]
⇒ `(setf medium-background)` *design medium* [*Generic Function*]

Sets the foreground and background ink, respectively, for the medium *medium* to *design*. You may not set `medium-foreground` or `medium-background` to an indirect ink.

design is an unbounded design. If the background design is not completely opaque at all points, the consequences are unspecified.

Changing the foreground or background of a sheet that supports output recording causes the contents of the stream's viewport to be erased and redrawn using the new foreground and background.

⇒ `medium-ink` *medium* [*Generic Function*]

The current drawing ink for the medium *medium*, which can be any design. The drawing functions draw with the color and pattern that this specifies. See Chapter 13 for a more complete description of inks. The `:ink` drawing option temporarily changes the value of `medium-ink`.

⇒ `(setf medium-ink)` *design medium* [*Generic Function*]

Sets the current drawing ink for the medium *medium* to *design*. *design* is as for `medium-foreground`, and may be an indirect ink as well.

⇒ `medium-transformation` *medium* [*Generic Function*]

The current user transformation for the medium *medium*. This transformation is used to transform the coordinates supplied as arguments to drawing functions to the coordinate system of the drawing plane. See Chapter 5 for a complete description of transformations. The `:transformation` drawing option temporarily changes the value of `medium-transformation`.

⇒ `(setf medium-transformation)` *transformation medium* [*Generic Function*]

Sets the current user transformation for the medium *medium* to the transformation *transformation*.

⇒ `medium-clipping-region` *medium* [*Generic Function*]

The current clipping region for the medium *medium*. The drawing functions do not affect the drawing plane outside this region. The `:clipping-region` drawing option temporarily changes the value of `medium-clipping-region`.

⇒ `(setf medium-clipping-region)` *region medium* [*Generic Function*]

Sets the current clipping region for the medium *medium* to *region*. *region* must be a subclass of `area`. Furthermore, some implementations may signal an error if the clipping region is not a rectangle or a region set composed entirely of rectangles.

⇒ `medium-line-style` *medium* [*Generic Function*]

The current line style for the *medium medium*. The line and arc drawing functions render according to this line style. See Section 10.3 for a complete description of line styles. The `:line-style` drawing option temporarily changes the value of `medium-line-style`.

⇒ `(setf medium-line-style)` *line-style medium* [*Generic Function*]

Sets the current line style for the *medium medium* to the *line style line-style*.

⇒ `medium-default-text-style` *medium* [*Generic Function*]

The default text style for the *medium medium*. `medium-default-text-style` will return a fully specified text style, unlike `medium-text-style`, which may return a text style with null components. Any text styles that are not fully specified by the time they are used for rendering are merged against `medium-default-text-style` using `merge-text-styles`. See Chapter 11 for a complete description of text styles.

⇒ `(setf medium-default-text-style)` *text-style medium* [*Generic Function*]

Sets the default text style for the *medium medium* to the *text style text-style*. *text-style* must be a fully specified text style.

Some CLIM implementations may arrange to erase and redraw the output on an output recording stream when the default text style of the stream is changed. Implementations that do this must obey the proper vertical spacing for output streams, and must reformat tables, graphs, and so forth, as necessary. Because of the expense of this operation, CLIM implementations are not required to support this.

⇒ `medium-text-style` *medium* [*Generic Function*]

The current text style for the *medium medium*. The text drawing functions, including ordinary stream output, render text as directed by this text style merged against the default text style. This controls both graphical text (such as that drawn by `draw-text*`) and stream text (such as that written by `write-string`). See Chapter 11 for a complete description of text styles. The `:text-style` drawing option temporarily changes the value of `medium-text-style`.

⇒ `(setf medium-text-style)` *text-style medium* [*Generic Function*]

Sets the current text style for the *medium medium* to the *text style text-style*. *text-style* need not be a fully merged text style.

⇒ `medium-current-text-style` *medium* [*Generic Function*]

The current, fully merged text style for the *medium medium*. This is the text style that will be used when drawing text output, and is the result of merging `medium-text-style` against `medium-default-text-style`.

10.2 Drawing Option Binding Forms

⇒ `with-drawing-options` *(medium* `&rest` *drawing-options)* `&body` *body* [*Macro*]

Binds the state of the *medium* designated by *medium* to correspond to the supplied drawing options, and executes the body with the new drawing options specified by *drawing-options* in effect. Each option causes binding of the corresponding component of the medium for the dynamic extent of the body. The drawing functions effectively do a `with-drawing-options` when drawing option arguments are supplied to them.

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

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

⇒ `invoke-with-drawing-options` *medium function* `&rest` *drawing-options* [*Generic Function*]

Binds the state of the *medium* *medium* to correspond to the supplied drawing options, and then calls the function *function* with the new drawing options in effect. *function* is a function of one argument, the medium; it has dynamic extent. *drawing-options* is a list of alternating keyword-value pairs, and must have even length. Each option in *drawing-options* causes binding of the corresponding component of the medium for the dynamic extent of the body.

medium can be a medium, a sheet that supports the sheet output protocol, or a stream that outputs to such a sheet. All classes that obey the medium protocol must implement a method for `invoke-with-drawing-options`.

The drawing options can be any of the following, plus any of the suboptions for line styles and text styles. The default value specified for a drawing option is the value to which the corresponding component of a medium is normally initialized.

⇒ `:ink` [*Option*]

A design that will be used as the ink for drawing operations. The drawing functions draw with the color and pattern that this specifies. The default value is `+foreground-ink+`. See Chapter 13 for a complete description of inks.

The `:ink` *ink* drawing option temporarily changes the value of `(medium-ink` *medium*`)` to *ink*, replacing the previous ink; the new and old inks are not combined in any way.

⇒ `:transformation` [*Option*]

This transforms the coordinates used as arguments to drawing functions to the coordinate system of the drawing plane. The default value is `+identity-transformation+`. See Chapter 5 for a complete description of transformations.

The `:transformation` *xform* drawing option temporarily changes the value of `(medium-transformation` *medium*`)` to `(compose-transformations (medium-transformation` *medium*`)` *xform*`)`.

⇒ `:clipping-region` [*Option*]

The drawing functions do not affect the drawing plane outside this region. The clipping region must be an `area`. Furthermore, some implementations might signal an error if the clipping region is not a rectangle or a region set composed entirely of rectangles. Rendering is clipped both by this clipping region and by other clipping regions associated with the mapping from the target drawing plane to the viewport that displays a portion of the drawing plane. The default is `+everywhere+`, or in other words, no clipping occurs in the drawing plane, only in the viewport.

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

Issue: A better explanation is needed. It does the right thing, but it's hard to tell that from this description. — DCPL

⇒ `:line-style` [*Option*]

The line and arc drawing functions render according to this line style. The line style suboptions and default are defined in Section 10.3.

The `:line-style` *ls* drawing option temporarily changes the value of `(medium-line-style` *medium*`)` to *ls*, replacing the previous line style; the new and old line styles are not combined in any way.

If line style suboptions are supplied, they temporarily change the value of `(medium-line-style` *medium*`)` to a line style constructed from the specified suboptions. Components not specified by suboptions are defaulted from the `:line-style` drawing option, if it is supplied, or else from the previous value of `(medium-line-style` *medium*`)`. That is, if both the `:line-style` option and line style suboptions are supplied, the suboptions take precedence over the components of the `:line-style` option.

⇒ `:text-style` [*Option*]

The text drawing functions, including ordinary stream output, render text as directed by this text style merged against the default text style. The default value has all null components. See Chapter 11 for a complete description of text styles, including the text style suboptions.

The `:text-style` *ts* drawing option temporarily changes the value of `(medium-text-style` *medium*`)` to `(merge-text-styles` *ts* `(medium-text-style` *medium*`))`.

If text style suboptions are supplied, they temporarily change the value of `(medium-text-style` *medium*`)` to a text style constructed from the specified suboptions, merged with the `:text-style` drawing option if it is specified, and then merged with the previous value of `(medium-text-style` *medium*`)`. That is, if both the `:text-style` option and text style suboptions are supplied, the suboptions take precedence over the components of the `:text-style` option.

10.2.1 Transformation "Convenience" Forms

The following three functions are no different than using `with-drawing-options` with the `:transformation` keyword argument supplied. However, they are sufficiently useful that they are provided as a convenience to programmers.

In order to preserve referential transparency, these three forms apply the translation, rotation, or scaling transformation first, then the rest of the transformation from `(medium-transformation` *medium*`)`. That is, the following two forms would return the same transformation (assuming that the medium's transformation in the second example is the identity transformation):

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

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

⇒ `with-translation` *(medium dx dy)* `&body` *body* [*Macro*]

Establishes a translation on the medium designated by *medium* that translates by *dx* in the x direction and *dy* in the y direction, and then executes *body* with that transformation in effect.

dx and *dy* are as for `make-translation-transformation`.

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

⇒ `with-scaling` *(medium sx* `&optional` *sy origin)* `&body` *body* [*Macro*]

Establishes a scaling transformation on the medium designated by *medium* that scales by *sx* in the x direction and *sy* in the y direction, and then executes *body* with that transformation in effect. If *sy* is not supplied, it defaults to *sx*. If *origin* is supplied, the scaling is about that point; if it is not supplied, it defaults to $(0, 0)$.

sx and *sy* are as for `make-scaling-transformation`.

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

⇒ `with-rotation` *(medium angle* `&optional` *origin)* `&body` *body* [*Macro*]

Establishes a rotation on the medium designated by *medium* that rotates by *angle*, and then executes *body* with that transformation in effect. If *origin* is supplied, the rotation is about that point; if it is not supplied, it defaults to $(0, 0)$.

angle and *origin* are as for `make-rotation-transformation`.

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

10.2.2 Establishing Local Coordinate Systems

⇒ `with-local-coordinates` *(medium* `&optional` *x y)* `&body` *body* [*Macro*]

Binds the dynamic environment to establish a local coordinate system on the *medium* designated by *medium* with the origin of the new coordinate system at the position (x, y). The "directionality" of the coordinate system is otherwise unchanged. *x* and *y* are real numbers, and both default to 0.

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

⇒ `with-first-quadrant-coordinates` *(medium* `&optional` *x y)* `&body` *body* [*Macro*]

Binds the dynamic environment to establish a local coordinate system on the *medium* designated by *medium* with the positive x axis extending to the right and the positive y axis extending upward, with the origin of the new coordinate system at the position (x, y). *x* and *y* are real numbers, and both default to 0.

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

10.3 Line Styles

A line or other path is a one-dimensional object. However in order to be visible, the rendering of a line must occupy some non-zero area on the display hardware. A *line style* represents the advice of CLIM to the rendering substrate on how to perform the rendering.

⇒ `line-style` [*Protocol Class*]

The protocol class for line styles. If you want to create a new class that behaves like a line style, it should be a subclass of `line-style`. Subclasses of `line-style` must obey the line style protocol.

⇒ `line-style-p` *object* [*Function*]

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

⇒ `standard-line-style` [*Class*]

An instantiable class that implements line styles. A subclass of `line-style`. This is the class that make-line-style instantiates. Members of this class are immutable.

⇒ make-line-style &key *unit thickness joint-shape cap-shape dashes* [*Function*]

Returns an object of class standard-line-style with the supplied characteristics. The arguments and their default values are described in Section 10.3.1.

10.3.1 Line Style Protocol and Line Style Suboptions

Each of these suboptions has a corresponding reader that can be used to extract a particular component from a line style. The generic functions decribed below comprise the line style protocol; all subclasses of line-style must implement methods for these generic functions.

⇒ :line-unit [*Option*]
⇒ line-style-unit *line-style* [*Generic Function*]

Gives the unit used for measuring line thickness and dash pattern length for the line style. Possible values are :normal, :point, or :coordinate. The meaning of these options is:

- :normal—thicknesses and lengths are given in a relative measure in terms of the usual or "normal" line thickness. The normal line thickness is the thickness of the "comfortably visible thin line", [1] which is a property of the underlying rendering substrate. This is the default.
- :point—thicknesses and lengths are given in an absolute measure in terms of printer's points (approximately $1/72$ of an inch). [2]
- :coordinate—this means that the same units should be used for line thickness as are used for coordinates. In this case, the line thickness is scaled by the medium's current transformation, whereas :normal and :point do not scale the line thickness.

⇒ :line-thickness [*Option*]
⇒ line-style-thickness *line-style* [*Generic Function*]

The thickness, in the units indicated by line-style-unit, of the lines or arcs drawn by a drawing function. The thickness must be a real number. The default is 1, which combined with the default unit of :normal, means that the default line drawn is the "comfortably visible thin line".

⇒ :line-joint-shape [*Option*]
⇒ line-style-joint-shape *line-style* [*Generic Function*]

Specifies the shape of joints between segments of unfilled figures. The possible shapes are :miter, :bevel, :round, and :none; the default is :miter. Note that the joint shape is implemented by the host window system, so not all platforms will necessarily fully support it.

[1] In some window systems, the phrase "thinnest visible line" is used. This is not appropriate for CLIM, which intends to be device independent. (For instance, the thinnest visible line on a 400 d.p.i. laser printer is a function of the user's viewing distance from the paper.) Another attribute of a "normal" line is that its thickness should approximately match the stroke thickness of "normal" text, where again the exact measurements are the province of the rendering engine, not of CLIM.

[2] This measure was chosen so that CLIM implementors who interface CLIM to an underlying rendering engine (the window system) may legitimately choose to make it render as 1 pixel on current (1990) display devices.

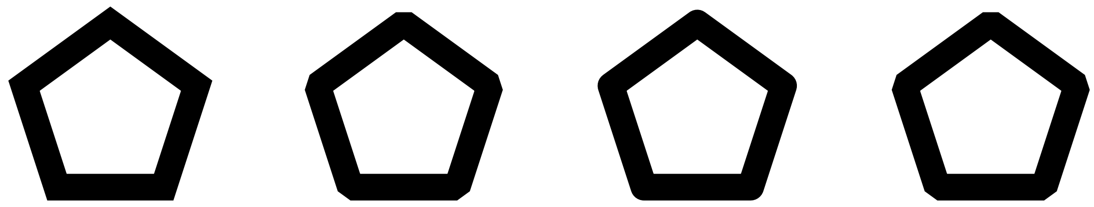


Figure 10.1: Line joint shapes.

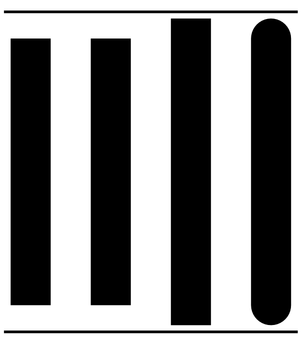


Figure 10.2: Line cap shapes.

⇒ `:line-cap-shape` [*Option*]
⇒ `line-style-cap-shape` *line-style* [*Generic Function*]

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

⇒ `:line-dashes` [*Option*]
⇒ `line-style-dashes` *line-style* [*Generic Function*]

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

- `nil`—lines are drawn solid, with no dashing. This is the default.
- `t`—lines are drawn dashed, with a dash pattern that is unspecified and may vary with the rendering

engine. This allows the underlying display substrate to provide a default dashed line for the programmer whose only requirement is to draw a line that is visually distinguishable from the default solid line.

- A sequence—specifies a sequence, usually a vector, controlling the dash pattern of a drawing function. It is an error if the sequence does not contain an even number of elements. The elements of the sequence are lengths (as real numbers) of individual components of the dashed line or arc. The odd elements specify the length of inked components, the even elements specify the gaps. All lengths are expressed in the units described by `line-style-unit`.

(See also `make-contrasting-dash-patterns`.)

10.3.2 Contrasting Dash Patterns

⇒ `make-contrasting-dash-patterns` *n* `&optional` *k* [*Function*]

If *k* is not supplied, this returns a vector of *n* dash patterns with recognizably different appearance. Elements of the vector are guaranteed to be acceptable values for `:dashes`, and do not include `nil`, but their class is not otherwise specified. The vector is a fresh object that may be modified.

If *k* is supplied, it must be an integer between 0 and $n-1$ (inclusive), in which case `make-contrasting-dash-patterns` returns the *k*'th dash-pattern rather than returning a vector of dash-patterns.

If the implementation does not have *n* different contrasting dash patterns, `make-contrasting-dash-patterns` signals an error. This will not happen unless *n* is greater than eight.

⇒ `contrasting-dash-pattern-limit` *port* [*Generic Function*]

Returns the number of contrasting dash patterns that can be rendered on any medium on the port *port*. Implementations are encouraged to make this as large as possible, but it must be at least 8. All classes that obey the port protocol must implement a method for this generic function.

Chapter 11

Text Styles

When specifying a particular "appearance" for rendered characters, there is a tension between portability and access to the specific device font. CLIM provides a portable mechanism for describing the desired *text style* in abstract terms. Each CLIM "port" defines a mapping between these abstract style specifications and particular device-specific fonts. In this way, an application programmer can specify the desired text style in abstract terms secure in the knowledge that an appropriate device font will be selected at run time by CLIM. However, some programmers may require direct access to particular device fonts. The text style mechanism supports specifying device fonts by name, allowing the programmer to sacrifice portability for control.

11.1 Text Styles

Text style objects have components for family, face, and size. Not all of these attributes need be supplied for a given text style object. Text styles can be merged in much the same way as pathnames are merged; unspecified components in the style object (that is, components that have `nil` in them) may be filled in by the components of a "default" style object. A text style object is called *fully specified* if none of its components is `nil`, and the size component is not a relative size (that is, is neither `:smaller` nor `:larger`).

⇒ `text-style` [*Protocol Class*]

The protocol class for text styles. If you want to create a new class that behaves like a text style, it should be a subclass of `text-style`. Subclasses of `text-style` must obey the text style protocol.

⇒ `text-style-p` *object* [*Function*]

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

⇒ `standard-text-style` [*Class*]

An instantiable class that implements text styles. It is a subclass of `text-style`. This is the class that `make-text-style` instantiates. Members of this class are immutable.

The interface to text styles is as follows:

⇒ `make-text-style` *family face size* [*Function*]

Returns an object of class `standard-text-style` with a family of *family*, a face of *face*, and a size of *size*.

family is one of `:fix`, `:serif`, `:sans-serif`, or `nil`.

face is one of `:roman`, `:bold`, `:italic`, `(:bold :italic)`, or `nil`.

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

Issue: Need to describe what family, face, size mean in terms of visual appearance. — York

⇒ `*default-text-style*` [*Constant*]

The default text style used by all CLIM mediums, if no text style was explicitly specified for the mediums. This must be a fully merged text style.

11.1.1 Text Style Protocol and Text Style Suboptions

The following generic functions comprise the text style protocol. All subclasses of `text-style` must implement methods for each of these generic functions.

Each of the suboptions described below has a corresponding reader accessor that can be used to extract a particular component from a text style.

⇒ `text-style-components` *text-style* [*Generic Function*]

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

⇒ `:text-family` [*Option*]
⇒ `text-style-family` *text-style* [*Generic Function*]

Specifies the family of the *text style text-style*.

⇒ `:text-face` [*Option*]
⇒ `text-style-face` *text-style* [*Generic Function*]

Specifies the face of the *text style text-style*.

⇒ `:text-size` [*Option*]
⇒ `text-style-size` *text-style* [*Generic Function*]

Specifies the size of the *text style* *text-style*.

⇒ `parse-text-style` *style-spec* [*Function*]

Returns a text style object. *style-spec* may be a `text-style` object or a device font, in which case it is returned as is, or it may be a list of the family, face, and size (that is, a "style spec"), in which case it is "parsed" and a `text-style` object is returned. This function is for efficiency, since a number of common functions that take a style object as an argument can also take a style spec, in particular `draw-text`.

⇒ `merge-text-styles` *style1 style2* [*Generic Function*]

Merges the *text styles* *style1* with *style2*, that is, returns a new text style that is the same as *style1*, except that unspecified components in *style1* are filled in from *style2*. For convenience, the two arguments may be also be style specs.

When merging the sizes of two text styles, if the size from *style1* is a relative size, the resulting size is either the next smaller or next larger size than is specified by *style2*. The ordering of sizes, from smallest to largest, is `:tiny`, `:very-small`, `:small`, `:normal`, `:large`, `:very-large`, and `:huge`.

Issue: Need to describe face-merging properly. — SWM

⇒ `text-style-ascent` *text-style port* [*Generic Function*]
⇒ `text-style-descent` *text-style port* [*Generic Function*]
⇒ `text-style-height` *text-style port* [*Generic Function*]
⇒ `text-style-width` *text-style port* [*Generic Function*]

Returns the ascent, descent, height, and width (respectively) of the font corresponding to the *text style* *text-style* as it would be rendered on any medium on the *port* *port*. *text-style* must be a fully specified text style.

The ascent of a font is the distance between the top of the tallest character in that font and the font's baseline. The descent of a font is the distance between the baseline and the bottom of the lowest descending character (usually "g", "p", "q", or "y"). The height of a font is the sum of the ascent and the descent of the font. The width of a font is the width of some representative character in the font.

The methods for these generic functions will typically specialize both the *text-style* and *port* arguments. Implementations should also provide "trampoline" for these generic function for mediums and output sheets; the trampolines will simply call the method for the port.

⇒ `text-style-fixed-width-p` *text-style port* [*Generic Function*]

Returns *true* if the *text styles* *text-style* will map to a fixed-width font on any medium on the *port* *port*, otherwise returns *false*. *text-style* must be a fully specified text style.

The methods for this generic function will typically specialize both the *text-style* and *port* arguments. Implementations should also provide a "trampoline" for this generic function for mediums and output sheets; the trampoline will simply call the method for the port.

Issue: Discuss baselines? Kerning? — SWM

⇒ `text-size` *medium string* `&key` *text-style (start* `0`*) end* [*Generic Function*]

Computes the "cursor motion" in device units that would take place if *string* (which may be either a string or a character) were output to the medium *medium* starting at the position $(0,0)$. Five values are returned: the total width of the string in device units, the total height of the string in device units, the final x cursor position (which is the same as the width if there are no `#\Newline` characters in the string), the final y cursor position (which is 0 if the string has no `#\Newline` characters in it, and is incremented by the line height of *medium* for each `#\Newline` character in the string), and the string's baseline.

text-style specifies what text style is to be used when doing the output, and defaults to `medium-merged-text-style` of the medium. *text-style* must be a fully specified text style. *start* and *end* may be used to specify a substring of *string*.

If a programmer needs to account for kerning or the ascent or descent of the text style, he should measure the size of the bounding rectangle of the text rendered on *medium*.

All mediums and output sheets must implement a method for this generic function.

11.2 Text Style Binding Forms

⇒ `with-text-style` *(medium text-style)* `&body` *body* [*Macro*]

Binds the current text style of the medium designated by *medium* to correspond to the new text style. *text-style* may either a text style object or a style spec (that is, a list of the a family, a face code, and a size). *body* is executed with the new text style in effect.

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

`with-text-style` must be implemented by expanding into a call to `invoke-with-text-style`, supplying a function that executes *body* as the *function* argument to `invoke-with-text-style`.

⇒ `invoke-with-text-style` *medium function text-style* [*Generic Function*]

Binds the current text style of the medium *medium* to correspond to the new text style, and calls the function *function* with the new text style in effect. *text-style* may either a text style object or a style spec (that is, a list of the a family, a face code, and a size). *function* is a function of one argument, the medium; it has dynamic extent.

medium can be a medium, a sheet that supports the sheet output protocol, or a stream that outputs to such a sheet. All classes that obey the medium protocol must implement a method for `invoke-with-text-style`.

⇒ `with-text-family` *(medium family)* `&body` *body* [*Macro*]
⇒ `with-text-face` *(medium face)* `&body` *body* [*Macro*]
⇒ `with-text-size` *(medium size)* `&body` *body* [*Macro*]

Binds the current text style of the *medium* designated by *medium* to correspond to a new text style consisting of the current text style with the new family, face, or size (respectively) merged in. *face*, *family*, and *size* are as for `make-text-style`. *body* is executed with the new text style in effect.

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

These macros are "convenience" forms of `with-text-style` that must expand into calls to `invoke-with-text-style`.

11.3 Controlling Text Style Mappings

Text styles are mapped to fonts using the `text-style-mapping` function, which takes a port, a character set, and a text style and returns a font object. All ports must implement methods for the following generic functions, for all classes of text style.

The objects used to represent a font mapping are unspecified and are likely to vary from port to port. For instance, a mapping might be some sort of font object on one type of port, or might simply be the name of a font on another.

Issue: We still need to describe what a device font is. Ditto, character sets. — SWM

Part of initializing a port is to define the mappings between text styles and font names for the port's host window system.

⇒ `text-style-mapping` *port character-set text-style* [*Generic Function*]

Returns the font mapping that will be used when rendering characters in the character set *character-set* in the *text style* *text-style* on any medium on the *port* *port*. If there is no mapping associated with *character-set* and *text-style* on *port*, then some other object will be returned that corresponds to the "unmapped" text style.

⇒ `(setf text-style-mapping)` *mapping port character-set text-style* [*Generic Function*]

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

Issue: While there is nothing wrong with this on the surface, there are indications that it is not really right. It turns out that several platforms don't even use this mechanism, which may indicate that it is not good enough. — SWM

⇒ `make-device-font-text-style` *display-device device-font-name* [*Function*]

Returns a text style object that will be mapped directly to the specified device font when text is output to a to the display device with this style. Device font styles do not merge with any other kind of style.

Chapter 12

Graphics

The CLIM graphic drawing model is an idealized model of graphical pictures. The model provides the language that application programs use to describe the intended visual appearance of textual and graphical output. Usually not all of the contents of the screen are described using the graphic drawing model. For example, menus and scroll bars might be described in higher-level terms.

An important aspect of the CLIM graphic drawing model is its extreme device independence. The model describes ideal graphical images and ignores limitations of actual graphics devices. One consequence of this is that the actual visual appearance of the screen can only be an approximation of the appearance specified by the model. Another consequence of this is that the model is highly portable.

CLIM separates output into two layers, a text/graphics layer in which one specifies the desired visual appearance independent of device resolution and characteristics, and a rendering layer in which some approximation of the desired visual appearance is created on the device. Of course application programs can inquire about the device resolution and characteristics if they wish and modify their desired visual appearance on that basis. (There is also a third layer above these two layers, the adaptive toolkit layer where one specifies the desired functionality rather than the desired visual appearance.)

Issue: There are still no functions to ask about device resolution and characteristics. Also, do we really need to refer to the adaptive toolkit layer here? — SWM

CLIM's drawing functions provide convenient ways to draw several commonly-used shapes.

The interaction between graphics and output recording will be described in Chapter 16.

12.1 Definitions

Drawing plane. A drawing plane is an infinite two-dimensional plane on which graphical output occurs. The drawing plane contains an arrangement of colors and opacities that is modified by each graphical output operation. It is not possible to read back the contents of a drawing plane, except by examining the output-history. Normally each window has its own drawing plane.

Coordinates. Coordinates are a pair of real numbers in implementation-defined units that identify a point in the drawing plane.

Sheets and Mediums. In this chapter, we use a sheet as a destination for output. Associated with a sheet is a medium, which has a drawing plane, two designs called the medium's foreground and background, a transformation, a clipping region, a line style, and a text style. There are per-medium, dynamically scoped, default drawing options. Different medium classes are provided to allow programmers to draw on different sorts of devices, such as displays, printers, and virtual devices such as bitmaps.

Note that a stream is a special kind of sheet that implements the stream protocol; streams include additional state such as the current text cursor (which is some point in the drawing plane). While the functions defined here are specified to be called on sheets, they can also be called on streams and mediums.

By default, the "fundamental" coordinate system of a CLIM stream (not a general sheet or medium, whose fundamental coordinate system is not defined) is a left handed system with x increasing to the right, and y increasing downward. $(0, 0)$ is at the upper left corner.

12.2 Drawing is Approximate

Note that although the drawing plane contains an infinite number of mathematical points, and drawing can be described as an infinite number of color and opacity computations, the drawing plane cannot be viewed directly and has no material existence. It is only an abstraction. What can be viewed directly is the result of rendering portions of the drawing plane onto a medium. No infinite computations or objects of infinite size are required to implement CLIM, because the results of rendering have finite size and finite resolution.

A drawing plane is described as having infinitely fine spatial, color, and opacity resolution, and as allowing coordinates of unbounded positive or negative magnitude. A viewport into a drawing plane, on the other hand, views only a finite region (usually rectangular) of the drawing plane. Furthermore, a viewport has limited spatial resolution and can only produce a limited number of colors. These limitations are imposed by the display hardware on which the viewport is displayed. A viewport also has limited opacity resolution, determined by the finite arithmetic used in the drawing engine (which may be hardware or software or both).

Coordinates are real numbers in implementation-defined units. Often these units equal the spatial resolution of a viewport, so that a line of thickness 1 is equivalent to the thinnest visible line. However, this equivalence is not required and should not be assumed by application programs.

A valid CLIM implementation can be quite restrictive in the size and resolution of its viewports. For example, the spatial resolution might be only a few dozen points per inch, the maximum size might be only a few hundred points on a side, and there could be as few as two displayable colors (usually black and white). The fully transparent and fully opaque opacity levels must always be supported, but a valid CLIM implementation might support only a few opacity levels in between (or possibly even none). A valid

CLIM implementation might implement color blending and unsaturated colors by stippling, although it is preferred, when possible, for a viewport to display a uniform color as a uniform color rather than as a perceptible stipple.

When CLIM records the output to a sheet, there are no such limitations since CLIM just remembers the drawing operations that were performed, not the results of rendering.

CLIM provides some ways to ask what resolution limits are in effect for a medium. See Chapter 10 for their descriptions.

The application programmer uses the CLIM graphic drawing model as an interface to describe the intended visual appearance. An implementation does its best to approximate that ideal appearance in a viewport, within its limitations of spatial resolution, color resolution, number of simultaneously displayable colors, and drawing speed. This will usually require tradeoffs, for example between speed and accuracy, and each implementation must make these tradeoffs according to its own hardware/software environment and user concerns. For example, if the actual device supports a limited number of colors, the desired color may be approximated by techniques such as dithering or stippling. If the actual device cannot draw curves exactly, they may be approximated, with or without anti-aliasing. If the actual device has limited opacity resolution, color blending may be approximate. A viewport might display colors that don't appear in the drawing plane, both because of color and opacity approximation and because of anti-aliasing at the edges of drawn shapes.

It is likely that different implementations will produce somewhat different visual appearance when running the same application. If an application requires more detailed control, it must resort to a lower-level interface, and will become less portable as a result. These lower-level interfaces will be documented on a per-platform basis.

Drawing computations are always carried out "in color", even if the viewport is only capable of displaying black and white. In other words, the CLIM drawing model is always the fully general model, even if an implementation's color resolution is limited enough that full use of the model is not possible. Of course an application that fundamentally depends on color will not work well on a viewport that cannot display color. Other applications will degrade gracefully.

Whether the implementation uses raster graphics or some other display technique is invisible at this interface. CLIM does not specify the existence of pixels nor the exact details of scan conversion, which will vary from one drawing engine to the next.

Performance will also vary between implementations. This interface is defined in terms of simple conceptual operations, however an actual implementation may use caching, specialized object representations, and other optimizations to avoid materializing storage-intensive or computation-costly intermediate results and to take advantage of available hardware.

12.3 Rendering Conventions for Geometric Shapes

The intent of this section is to describe the conventions for how CLIM should render a shape on a display device. These conventions and the accompanying examples are meant to describe a set of goals that a CLIM implementation should try to meet. However, compliant CLIM implementations may deviate from these goals if necessary (for example, if the rendering performance on a specific platform would be unacceptably slow if these goals were met exactly and implementors feel that users would be better served by speed than by accuracy). Note that we discuss only pixel-based display devices here, which are the most common, but by no means the only, sort of display device that can be supported by CLIM.

When CLIM draws a geometric shape on some sort of display device, the idealized geometric shape must somehow be rendered on the display device. The geometric shapes are made up of a set of mathematical points, which have no size; the rendering of the shape is usually composed of pixels, which are roughly square. These pixels exist in "device coordinates", which are gotten by transforming the user-supplied coordinates by all of the user-supplied transformation, the medium transformation, and the transformation that maps from the sheet to the display device. (Note that if the last transformation is a pure translation that translates by an integer multiple of device units, then it has no effect on the rendering other than placement of the figure drawn on the display device.)

Roughly speaking, a pixel is affected by drawing a shape only when it is inside the shape (we will define what we mean by "inside" in a moment). Since pixels are little squares and the abstract points have no size, for most shapes there will be many pixels that lie only partially inside the shape. Therefore, it is important to describe the conventions used by CLIM as to which pixels should be affected when drawing a shape, so that the proper interface to the per-platform rendering engine can be constructed. (It is worth noting that on devices that support color or grayscale, the rendering engine may attempt to draw a pixel that is partially inside the shape darker or lighter, depending on how much of it is inside the shape. This is called *anti-aliasing*.) The conventions used by CLIM is the same as the conventions used by X11:

- A pixel is a addressed by its upper-left corner.
- A pixel is considered to be *inside* a shape, and hence affected by the rendering of that shape, if the center of the pixel is inside the shape. If the center of the pixel lies exactly on the boundary of the shape, it is considered to be inside if the inside of the shape is immediately to the right (increasing x direction on the display device) of the center point of the pixel. If the center of the pixel lies exactly on a horizontal boundary, it is considered to be inside if the inside of the shape is immediately below (increasing y direction on the display device) the center point of the pixel.
- An unfilled shape is drawn by taking the filled shape consisting of those points that are within 1/2 the line thickness from the outline curve (using a normal distance function, that is, the length of the line drawn at right angles to the tangent to the outline curve at the nearest point), and applying the second rule, above.

It is important to note that these rules imply that the decision point used for insideness checking is offset from the point used for addressing the pixel by half a device unit in both the x and y directions. It is worth considering the motivations for these conventions.

When two shapes share a common edge, it is important that only one of the shapes own any pixel. The two triangles in Figure 12.1 illustrate this. The pixels along the diagonal belong to the lower figure. When the decision point of the pixel (its center) lies to one side of the line or the other, there is no issue. When the boundary passes through a decision point, which side the inside of the figure is on is used to decide. These are the triangles that CLIM implementations should attempt to draw in this case.

The reason for choosing the decision point half a pixel offset from the address point is to reduce the number of common figures (such as rectilinear lines and rectangles with integral coordinates) that invoke

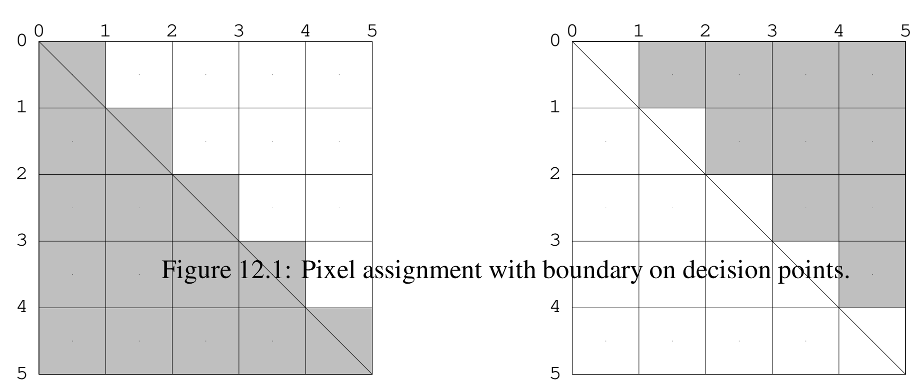


Figure 12.1: Pixel assignment with boundary on decision points.

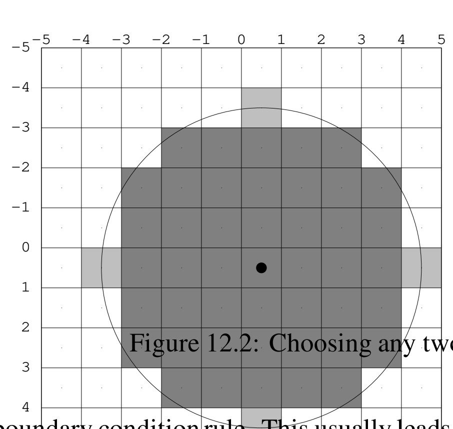


Figure 12.2: Choosing any two of the shaded pixels causes asymmetry.

the boundary condition rule. This usually leads to more symmetrical results. For instance, in Figure 12.2, we see a circle drawn when the decision point is the same as the address point. The four lighter points are indeterminate: it is not clear whether they are inside or outside the shape. Since we want to have each boundary case determined according to which side has the figure on it, and since we must apply the same rule uniformly for all figures, we have no choice but to pick only two of the four points, leading to an undesirable lopsided figure.

If we had instead chosen to take all four boundary points, we would have a nice symmetrical figure. However, since this figure is symmetrical about a whole pixel, it is one pixel wider than it ought to be. The problem with this can be seen clearly in Figure 12.3 if we attempt to draw a rectangle and circle overlaid with the following code:

```
(defun draw-test (sheet radius)
  (draw-circle* sheet 0 0 radius :ink +foreground-ink+)
  (draw-rectangle* sheet (- radius) (- radius) (+ radius) (+ radius)
```

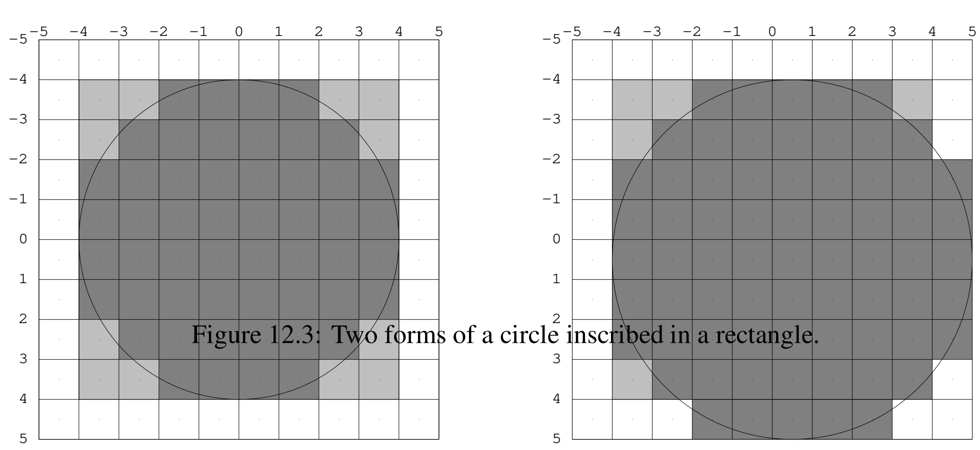


Figure 12.3: Two forms of a circle inscribed in a rectangle.

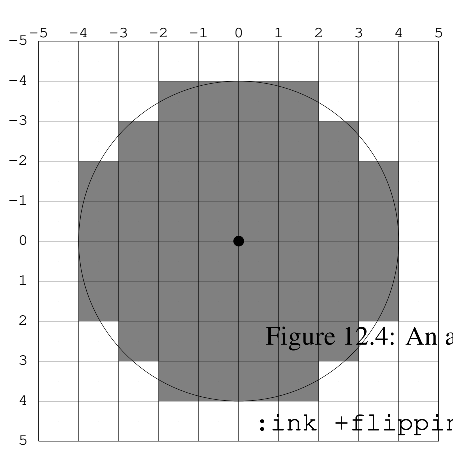


Figure 12.4: An aesthetically pleasing circle.

```
:ink +flipping-ink+))
```

It is for this reason that we choose to have the decision point at the center of the pixel. This draws circles that look like the one in Figure 12.4. It is this shape that CLIM implementations should attempt to draw.

A consequence of these rendering conventions is that, when the start or end coordinate (minus 1/2 the line thickness, if the shape is a path) is not an integer, then rendering is not symmetric under reflection transformations. Thus to correctly and portably draw an outline of thickness 1 around a (rectilinear) rectangular area with integral coordinates, the outline path must have half-integral coordinates. Drawing rectilinear areas whose boundaries are not on pixel boundaries cannot be guaranteed to be portable. Another way to say the same thing is that the “control points” for a rectangular area are at the corners, while the control points for a rectilinear path are in the center of the path, not at the corners. Therefore, in order for a path and an area to abut seamlessly, the coordinates of the path must be offset from the coordinates of the area by half the path’s thickness.

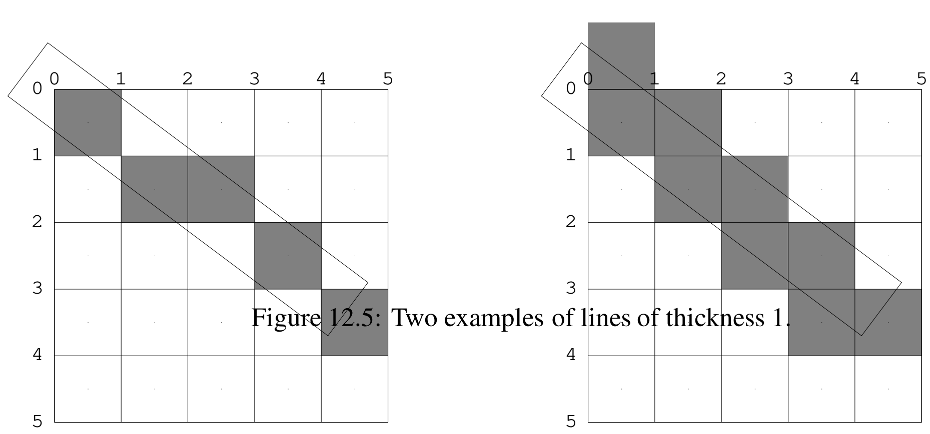


Figure 12.5: Two examples of lines of thickness 1.

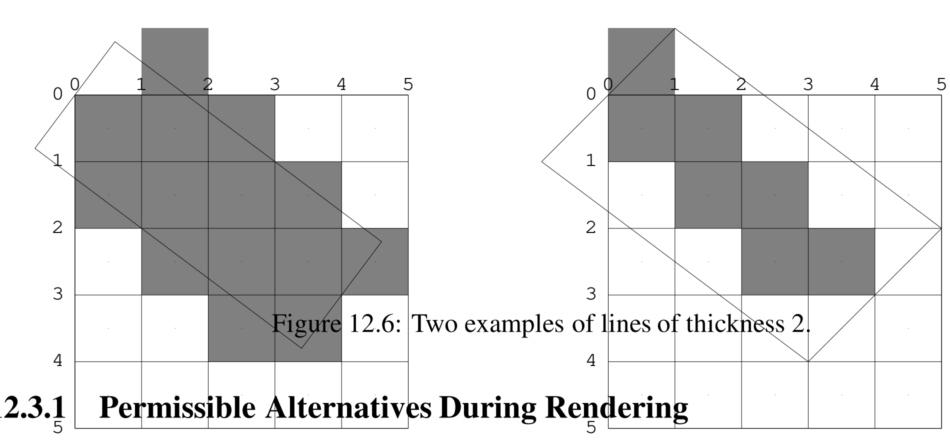


Figure 12.6: Two examples of lines of thickness 2.

12.3.1 Permissible Alternatives During Rendering

Some platforms may distinguish between lines of the minimum thinness from lines that are thicker than that. The two rasterizations depicted in Figure 12.5 are both perfectly reasonable rasterizations of tilted lines that are a single device unit wide. The right-hand line is drawn as a tilted rectangle, the left as the "thinnest visible" line.

For thick lines, a platform may choose to draw the exact tilted fractional rectangle, or the coordinates of that rectangle might be rounded so that it is distorted into another polygonal shape. The latter case may be prove to be faster on some platforms. The two rasterizations depicted in Figure 12.6 are both reasonable.

The decision about which side of the shape to take when a boundary line passes through the decision point is made arbitrarily, although we have chosen to be compatible with the X11 definition. This is not necessarily the most convenient decision. The main problem with this is illustrated by the case of a horizontal line (see Figure 12.7). Our definition chooses to draw the rectangular slice above the

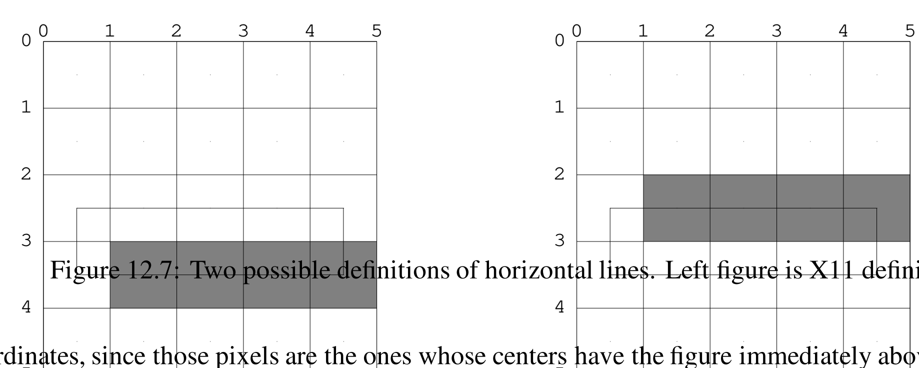


Figure 12.7: Two possible definitions of horizontal lines. Left figure is X11 definition.

coordinates, since those pixels are the ones whose centers have the figure immediately above them. This definition makes it simpler to draw rectilinear borders around rectilinear areas.

12.4 Drawing Functions

Each drawing function takes keyword arguments allowing any drawing option or suboption to be supplied separately in the call to the function. In some implementations of CLIM, the drawing functions may ignore drawing options that are irrelevant to that function; in other implementations, an error may be signalled. See Chapter 10 for a more complete discussion of the drawing options. An error will be signalled if any drawing function is called on a sheet that is mute for output.

While the functions in this section are specified to be called on sheets, they can also be called on streams and mediums.

Implementation note: The drawing functions are all specified as ordinary functions, not as generic functions. This is intended to ease the task of writing compile-time optimizations that avoid keyword argument taking, check for such things as constant drawing options, and so forth. If you need to specialize any of the drawing methods, use `define-drawing-function-method`.

Each drawing function comes in two forms, a "structured" version and a "spread" version. The structured version passes points, whereas the spread version passes coordinates. See Section 2.3 for more information on this.

Any drawing functions may create an *output record* that corresponds to the figure being drawn. See Chapter 15 for a complete discussion of output recording. During output recording, none of these functions capture any arguments that are points, point sequences, coordinate sequences, or text strings. Line styles, text styles, transformations, and clipping regions may be captured.

Note that the CLIM specification does not specify more complex shapes such as cubic splines and Bézier curves. These are suitable candidates for extensions to CLIM.

⇒ `draw-point` *sheet point* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]
⇒ `draw-point*` *sheet x y* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]

These functions (structured and spread arguments, respectively) draw a single point on the sheet *sheet* at the point *point* (or the position (x, y)).

The unit and thickness components of the current line style (see Chapter 10) affect the drawing of the point by controlling the size on the display device of the "blob" that is used to render the point.

⇒ `draw-points` *sheet point-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]
⇒ `draw-points*` *sheet coord-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]

These functions (structured and spread arguments, respectively) draw a set of points on the sheet *sheet*. *point-seq* is a sequence of point objects; *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements.

These functions exist as equivalents to

```
(map nil #'(lambda (point) (draw-point sheet point)) point-seq)
```

and

```
(do ((i 0 (+ i 2)))
    ((= i (length coord-seq)))
  (draw-point* sheet (elt coord-seq i) (elt coord-seq (+ i 1))))
```

for convenience and efficiency.

⇒ `draw-line` *sheet point1 point2* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-line*` *sheet x1 y1 x2 y2* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a line segment on the sheet *sheet* from the point *point1* to *point2* (or from the position (*x1*,*y1*) to (*x2*,*y2*)).

The current line style (see Chapter 10) affects the drawing of the line in the obvious way, except that the joint shape has no effect. Dashed lines start dashing at *point1*.

⇒ `draw-lines` *sheet point-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-lines*` *sheet coord-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a set of disconnected line segments. *point-seq* is a sequence of point objects; *coord-seq* is a sequence of coordinate pairs. It is an error if *coord-seq* does not contain an even number of elements.

These functions are equivalent to

```
(do ((i 0 (+ i 2)))
    ((= i (length point-seq)))
  (draw-line sheet (elt point-seq i) (elt point-seq (1+ i))))
```

and

```
(do ((i 0 (+ i 4)))
    ((= i (length coord-seq)))
  (draw-line* sheet (elt coord-seq i) (elt coord-seq (+ i 1))
                    (elt coord-seq (+ i 2)) (elt coord-seq (+ i 3))))
```

⇒ `draw-polygon` *sheet point-seq* `&key` *(filled* `t`*) (closed* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape* [*Function*]
⇒ `draw-polygon*` *sheet coord-seq* `&key` *(filled* `t`*) (closed* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape* [*Function*]

Draws a polygon or polyline on the sheet *sheet*. When *filled* is false, this draws a set of connected lines, otherwise it draws a filled polygon. If *closed* is true (the default) and *filled* is false, it ensures that a segment is drawn that connects the ending point of the last segment to the starting point of the first segment. The current line style (see Chapter 10) affects the drawing of unfilled polygons in the obvious way. The cap shape affects only the "open" vertices in the case when *closed* is false. Dashed lines start dashing at the starting point of the first segment, and may or may not continue dashing across vertices, depending on the window system.

point-seq is a sequence of point objects; *coord-seq* is a sequence of coordinate pairs. It is an error if *coord-seq* does not contain an even number of elements.

If *filled* is true, a closed polygon is drawn and filled in. In this case, *closed* is assumed to be true as well.

⇒ `draw-rectangle*` *sheet x1 y1 x2 y2* `&key` *(filled* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape* [*Function*]

Draws either a filled or unfilled rectangle on the sheet *sheet* that has its sides aligned with the coordinate axes. One corner of the rectangle is at the position (*x1*,*y1*) and the opposite corner is at (*x2*,*y2*). The arguments *x1*, *y1*, *x2*, and *y1* are real numbers that are canonicalized in the same way as for `make-bounding-rectangle`. *filled* is as for `draw-polygon*`.

Issue: This needs some good short text to explain that "aligned with the coordinate axes" is the axes relative to the current transformation, not the result of transforming the points. But I'm having trouble *dreaming up such text. — DCPL*

The current line style (see Chapter 10) affects the drawing of unfilled rectangles in the obvious way, except that the cap shape has no effect.

⇒ `draw-ellipse` *sheet center-pt radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

⇒ `draw-ellipse*` *sheet center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw an ellipse (when *filled* is true, the default) or an elliptical arc (when *filled* is false) on the sheet *sheet*. The center of the ellipse is the point *center-pt* (or the position (*center-x*,*center-y*)).

Two vectors, (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*) specify the bounding parallelogram of the ellipse as explained in Chapter 3. All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the `ellipse-not-well-defined` error will be signalled. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. Angles are measured with respect to the positive x axis. The elliptical arc runs positively (counter-clockwise) from *start-angle* to *end-angle*. The default for *start-angle* is 0; the default for *end-angle* is 2π.

In the case of a "filled arc" (that is when *filled* is true and *start-angle* or *end-angle* are supplied and are not 0 and 2π), the figure drawn is the "pie slice" area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from *start-angle* to *end-angle*.

When drawing unfilled ellipses, the current line style (see Chapter 10) affects the drawing in the obvious way, except that the joint shape has no effect. Dashed elliptical arcs start "dashing" at *start-angle*.

⇒ `draw-circle` *sheet center-pt radius* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

⇒ `draw-circle*` *sheet center-x center-y radius* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a circle (when *filled* is true, the default) or a circular arc (when *filled* is false) on the sheet *sheet*. The center of the circle is *center-pt* or (*center-x*,*center-y*) and the radius is *radius*. These are just special cases of `draw-ellipse` and `draw-ellipse*`. *filled* is as for `draw-ellipse*`.

start-angle and *end-angle* allow the specification of an arc rather than a complete circle in the same manner as that of the ellipse functions, above.

The "filled arc" behavior is the same as that of an ellipse, above.

⇒ `draw-text` *sheet string-or-char point* `&key` *text-style (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*)* *toward-point transform-glyphs ink clipping-region transformation text-style text-family text-face text-size* [*Function*]

⇒ `draw-text*` *sheet string-or-char x y* `&key` *text-style (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*) toward-x toward-y transform-glyphs ink clipping-region transformation text-style text-family text-face text-size* [*Function*]

The text specified by *string-or-char* is drawn on the sheet *sheet* starting at the position specified by the point *point* (or the position (x, y)). The exact definition of "starting at" is dependent on *align-x* and *align-y*. *align-x* is one of `:left`, `:center`, or `:right`. *align-y* is one of `:baseline`, `:top`, `:center`, or `:bottom`. *align-x* defaults to `:left` and *align-y* defaults to `:baseline`; with these defaults, the first glyph is drawn with its left edge and its baseline at *point*.

text-style defaults to `nil`, meaning that the text will be drawn using the current text style of the sheet's medium.

start and *end* specify the start and end of the string, in the case where *string-or-char* is a string. If *start* is supplied, it must be an integer that is less than the length of the string. If *end* is supplied, it must be an integer that is less than the length of the string, but greater than or equal to *start*.

Normally, glyphs are drawn from left to right no matter what transformation is in effect. *toward-x* or *toward-y* (derived from *toward-point* in the case of `draw-text`) can be used to change the direction from one glyph to the next one. For example, if *toward-x* is less than the x position of *point*, then the glyphs will be drawn from right to left. If *toward-y* is greater than the y position of *point*, then the glyphs' baselines will be positioned one above another. More precisely, the reference point in each glyph lies on a line from *point* to *toward-point*, and the spacing of each glyph is determined by packing rectangles along that line, where each rectangle is "char-width" wide and "char-height" high.

If *transform-glyphs* is true, then each glyph is transformed as an image before it is drawn. That is, if a rotation transformation is in effect, then each glyph will be rotated individually. If *transform-glyphs* is not supplied, then the individual glyphs are not subject to the current transformation. It is permissible for CLIM implementations to ignore *transform-glyphs* if it is too expensive to implement.

12.5 Pixmaps

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

The exact representation of a pixmap is explicitly unspecified. There is no interaction between the pixmap operations and output recording, that is, displaying a pixmap on a sheet is a pure drawing operation that affects only the display, not the output history. Some sheets may not support pixmaps; in this case, an error will be signalled.

⇒ `allocate-pixmap` *sheet width height* [*Generic Function*]

Allocates and returns a pixmap object that can be used on any sheet that shares the same characteristics as *sheet*. The exact definition of "shared characteristics" will vary from host to host. The pixmap is *width* units wide, *height* units high, and as deep as is necessary to store the information for the sheet. The exact representation of pixmaps is explicitly unspecified.

The returned value is the pixmap.

⇒ `deallocate-pixmap` *pixmap* [*Generic Function*]

Deallocates the pixmap *pixmap*.

⇒ `copy-to-pixmap` *sheet sheet-x sheet-y width height* `&optional` *pixmap (pixmap-x* `0`*) (pixmap-y* `0`*)* [*Generic Function*]

Copies the pixels from the sheet *sheet* starting at the position specified by (*sheet-x*,sheet-y) into the pixmap *pixmap* at the position specified by (*pixmap-x*,*pixmap-y*). A rectangle whose width and height is specified by *width* and *height* is copied. *sheet-x* and *sheet-y* are specified in window coordinates, where $(0,0)$ is the upper-left corner of the window, including the margin area.

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

The returned value is the pixmap.

⇒ `copy-from-pixmap` *pixmap pixmap-x pixmap-y width height sheet window-x window-y* [*Generic Function*]

Copies the pixels from the pixmap *pixmap* starting at the position specified by (*pixmap-x*,*pixmap-y*) into the sheet *sheet* at the position (*sheet-x*,sheet-y). A rectangle whose width and height is specified by *width* and *height* is copied. *sheet-x* and *sheet-y* are specified in window coordinates, where $(0,0)$ is the upper-left corner of the window, including the margin area.

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

The returned value is the pixmap.

This is intended to specialize on both the *pixmap* and *sheet* arguments.

⇒ `copy-area` *sheet from-x from-y width height to-x to-y* [*Generic Function*]

Copies the pixels from the sheet *sheet* starting at the position specified by (*from-x*,*from-y*) to the position (*to-x*,*to-y*) on the same sheet. A rectangle whose width and height is specified by *width* and *height* is copied. *from-x*, *from-y*, *to-x*, and *to-y* are specified in window coordinates, where $(0,0)$ is the upper-left corner of the window, including the margin area.

⇒ `with-output-to-pixmap-stream` *(stream-var sheet)* `&body` *body* [*Macro*]

Binds *stream-var* to a "pixmap stream", that is, a stream that does output to a pixmap with the characteristics appropriate to the sheet *sheet*, and then evaluates *body* in that context. All the output done to the stream *stream-var* inside of *body* is drawn on the pixmap stream.

The returned value is a pixmap that can be drawn on *sheet* using `copy-from-pixmap`.

12.6 Graphics Protocols

Every medium must implement methods for the various graphical drawing generic functions. Furthermore, every sheet that supports the standard output protocol must implement these methods as well; often, the sheet methods will trampoline to the methods on the sheet's medium. All of these generic functions take the same arguments as the non-generic spread function equivalents, except the arguments that are keyword arguments in the non-generic functions are positional arguments in the generic functions.

Every port must implement methods for the various graphical drawing generic functions. All of these generic functions take as (specialized) arguments the port and medium, followed by an x and a y offset, followed by the drawing function-specific arguments, followed by the ink, line style (or text style), and clipping region.

The drawing function-specific arguments will either be x and y positions, or a sequence of x and y positions. These positions will be in medium coordinates, and must be transformed by applying the medium's device transformation in order to produce device coordinates. The x and y offsets are real numbers that can be used to optimize the implementation of a simple scrolling transformation. These offsets must be added to the positions before the device transformation is applied. Note that the user transformation will have already been applied to the positions when the port-specific drawing function is called.

The ink, line style (or text style), and clipping regions arguments are optional, and default from the medium (`medium-ink`, `medium-line-style` (or `medium-current-text-style`), and `medium-clipping-region`, respectively). Each port method will decode the ink, line (or text) style, and clipping region in a port-specific way and communicate it to the underlying port.

Issue: Do we need `define-graphics-generic` *and* `define-graphics-method`*, or whatever, so that the graphics protocol is easier to extend? Such things should also help define the hooks into output recording. — SWM*

12.6.1 General Behavior of Drawing Functions

Using `draw-line*` as an example, calling any of the drawing functions specified above results in the following series of function calls on a non-output recording sheet:

- A program calls `draw-line*` on arguments *sheet*, *x1*, *y1*, *x2*, and *y2*, and perhaps some drawing options.
- `draw-line*` merges the supplied drawing options into the sheet's medium, and then calls `medium-draw-line` on the sheet. (Note that a compiler macro could detect the case where there are no drawing options or constant drawing options, and do this at compile time.)

- `medium-draw-line` on the sheet trampolines to the medium's `medium-draw-line` method.
- `medium-draw-line` performs the necessary user transformations by applying the medium transformation to *x1*, *y1*, *x2*, and *y2*, and to the clipping region. Then `medium-draw-line` calls `port-draw-line` to do the actual drawing.

12.6.2 Medium-specific Drawing Functions

All mediums and all sheets that support the standard output protocol must implement methods for the following generic functions.

: Note: CLIM implementations should provide "trampoline" methods on sheets that support the standard output protocol that simply call the same generic function on the medium. Sheets that support output recording will require extra mechanism before delegating to the medium in order to implement such functionality as creating output records and handling scrolling.

⇒ `medium-draw-point` *medium x y* [*Generic Function*]

Draws a point on the medium *medium*. The point is drawn at (x, y), transformed by the medium's current transformation. `medium-draw-point` will call `port-draw-point` in order to draw the point.

⇒ `medium-draw-points` *medium coord-seq* [*Generic Function*]

Draws a set of points on the medium *medium*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. The coordinates in *coord-seq* are transformed by the medium's current transformation. `medium-draw-points` will call `port-draw-points` in order to draw the points.

⇒ `medium-draw-line` *medium x1 y1 x2 y2* [*Generic Function*]

Draws a line on the medium *medium*. The line is drawn from $(x1, y1)$ to $(x2, y2)$, with the start and end positions transformed by the medium's current transformation. `medium-draw-line` will call `port-draw-line` in order to draw the line.

⇒ `medium-draw-lines` *medium coord-seq* [*Generic Function*]

Draws a set of disconnected lines on the medium *medium*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. The coordinates in *coord-seq* are transformed by the medium's current transformation. `medium-draw-lines` will call `port-draw-lines` in order to draw the line.

⇒ `medium-draw-polygon` *medium coord-seq closed* [*Generic Function*]

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

⇒ `medium-draw-rectangle` *medium x1 y1 x2 y2* [*Generic Function*]

Draws a rectangle on the medium *medium*. The corners of the rectangle are at $(x1, y1)$ and $(x2, y2)$, with the corner positions transformed by the medium's current transformation. `medium-draw-rectangle` will call `port-draw-rectangle` in order to draw the rectangle.

⇒ `medium-draw-ellipse` *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle* [*Generic Function*]

Draws an rectangle on the medium *medium*. The center of the rectangle is at (x, y), and the radii are specified by the two vectors (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*). The center point and radii are transformed by the medium's current transformation. `medium-draw-ellipse` will call `port-draw-ellipse` in order to draw the ellipse.

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. The medium transformation must be applied to the angles as well.

⇒ `medium-draw-text` *medium text x y (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*) toward-x toward-y transform-glyphs* [*Generic Function*]

Draws a character or a string on the medium *medium*. The text is drawn starting at (x, y), and towards (*toward-x*,*toward-y*); these positions are transformed by the medium's current transformation. `medium-draw-text` will call either `port-draw-string` or `port-draw-character` in order to draw the text.

12.6.3 Port-specific Drawing Functions

The following port-specific functions are required by the drawing functions in the API. If the optional arguments to these functions are not supplied, they are defaulted from the medium.

⇒ `port-draw-point` *port medium x-offset y-offset x y* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws a point on the medium *medium* on the port *port*. The point is drawn at (x, y), with *x-offset* added to x and *y-offset* added to y, and the device transformation applied.

⇒ `port-draw-points` *port medium x-offset y-offset coord-seq* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws a set of points on the medium *medium* on the port *port*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. `port-draw-points` could be implemented by repeatedly calling port-draw-point, but some ports may support faster primitives to do this.

⇒ `port-draw-line` *port medium x-offset y-offset x1 y1 x2 y2* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws a line on the medium *medium* on the port *port*. The line is drawn from $(x1, y1)$ to $(x2, y2)$, with *x-offset* added to $x1$ and $x2$ and *y-offset* added to $y1$ and $y2$, and the device transformation applied.

⇒ `port-draw-lines` *port medium x-offset y-offset coord-seq* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws a set of disconnected lines on the medium *medium* on the port *port*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. `port-draw-lines` could be implemented by repeatedly calling port-draw-line, but some ports may support faster primitives to do this.

⇒ `port-draw-polygon` *port medium x-offset y-offset coord-seq closed* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws a polygon or polyline on the medium *medium* on the port *port*. When *line-style* is `nil`, this draws a filled polygon; otherwise, *line-style* is the line style to use when drawing a set of connected lines. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. The coordinate sequence specifies the corners of the polygon; *x-offset* is added to the x coordinates and *y-offset* is added to the y coordinates, and the device transformation is then applied.

When `port-draw-polygon` is drawing a set of connected lines, then if *closed* is true, a closed polygon is drawn. If *closed* is false, the polygon will not be closed.

⇒ `port-draw-rectangle` *port medium x-offset y-offset x1 y1 x2 y2* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws a rectangle on the medium *medium* on the port *port*. When *line-style* is `nil`, this draws a filled rectangle; otherwise, *line-style* is the line style to use when drawing an unfilled rectangle. The corners of the rectangle are at $(x1, y1)$ and $(x2, y2)$, with *x-offset* added to $x1$ and $x2$ and *y-offset* added to $y1$ and $y2$, and the device transformation applied.

⇒ `port-draw-ellipse` *port medium x-offset y-offset center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle* `&optional` *ink line-style clipping-region* [*Generic Function*]

Draws an ellipse rectangle on the medium *medium* on the port *port*. When *line-style* is `nil`, this draws a filled ellipse; otherwise, *line-style* is the line style to use when drawing an unfilled ellipse.

The center of the ellipse is at (*center-x*,*center-y*), with *x-offset* and *y-offset* added to *center-x* and *center-y*, and the device transformation applied. Two vectors, (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*) specify the bounding parallelogram of the ellipse as explained in Chapter 3. When the ellipse vectors have the same length, the ellipse is circle. The device transformation must be applied to the vectors as well.

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. The device transformation must be applied to the angles as well.

⇒ `port-draw-string` *port medium x-offset y-offset text x y (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*) toward-x toward-y transform-glyphs* `&optional` *ink text-style clipping-region* [*Generic Function*]

Draws a string on the *medium medium* on the *port port*. The string is drawn starting at (x, y), and towards (*toward-x*,*toward-y*). *align-x* is one of `:left`, `:center`, or `:right`. *align-y* is one of `:baseline`, `:top`, `:center`, or `:bottom`.

start and *end* specify the start and end of the string. *start* is an integer that is less than the length of the string. *end* is an integer that is less than the length of the string, but greater than or equal to *start*.

Normally, glyphs are drawn from left to right, but *toward-x* and *toward-y* can be used to change the direction from one glyph to the next one. The reference point in each glyph lies on a line from (x, y) to (*toward-x*,*toward-y*) and the spacing of each glyph is determined by packing rectangles along that line, where each rectangle is "char-width" wide and "char-height" high. Of course, some ports may not be able to fully support this.

If *transform-glyphs* is *true*, then each glyph is transformed as an image before it is drawn. It is permissible for CLIM implementations to ignore *transform-glyphs* if it is too expensive to implement.

⇒ `port-draw-character` *port medium x-offset y-offset character x y (align-x* `:left`*) (align-y* `:baseline`*)* `&optional` *ink text-style clipping-region* [*Generic Function*]

Draws a single character on the *medium medium* on the *port port* at the position (x, y). *align-x* is one of `:left`, `:center`, or `:right`. *align-y* is one of `:baseline`, `:top`, `:center`, or `:bottom`.

Chapter 13

Drawing in Color

This chapter describes the `:ink` drawing option and the simpler values that can be supplied for that option, such as colors. More complex values that have a regular or irregular pattern in the ink are described in Chapter 14.

Issue: We need to add a thing called a "raster ink", which includes things like plane masks, pixel values, etc. Be clear that this is a platform and device dependent. — SWM

Issue: We need to add a thing called a "palette", which is simply an abstract color map. — SWM

13.1 The `:ink` Drawing Option

The `:ink` drawing option, used with the drawing functions described in Chapter 12, can take as its value:

- a color,
- an opacity,
- the constant `+foreground-ink+`,
- the constant `+background-ink+`,
- a flipping ink, or
- other values described in Chapter 14

More exactly, an ink can be any member of the class `design`. For now you may think of a design as a possibly translucent color. More general designs are described in Chapter 14.

The drawing functions work by selecting a region of the drawing plane and painting it with color. The region to be painted is the intersection of the shape specified by the drawing function and the `:clipping-`

`region` drawing option, which is then transformed by the `:transformation` drawing option. The `:ink` drawing option is a design that specifies a new arrangement of colors (and opacities) in this region of the medium's drawing plane. Any viewports or dataports attached to this drawing plane are updated accordingly. The `:ink` drawing option is never affected by the `:transformation` drawing option nor by the sheet transformation; this ensures that stipple patterns on adjacent sheets join seamlessly.

Issue: The description of how the clipping region and transformations contribute isn't good enough. It is true if there are no other transformations and clipping regions present, and both are specified in the current drawing operation. But it doesn't say what happens if things are nested. I'm not sure it needs to. Rather, I think it should just say that the the region is clipped by the current clipping region in effect, then transformed by the current transform in effect, and that the rules for these are discussed in the drawing options section. — DCPL

Drawing consists conceptually of the following sequence of operations, performed in parallel at every point in the drawing plane. Of course, the actual implementation does not involve an infinite (or large parallel) computation.

1. The design specifies a color and an opacity at the point. These can depend on the drawing plane's current color and opacity, on the medium's foreground color, and on the medium's background color.
2. The color blending function is applied to the design's color and opacity and the drawing plane's color and opacity, returning a new color and opacity for the point.
3. The drawing plane's color and opacity at that point are set to the new color and opacity.

13.2 Designs

⇒ `design` [*Protocol Class*]

A design is an object that represents a way of arranging colors and opacities in the drawing plane. The `design` class is the protocol class for designs. If you want to create a new class that behaves like a design, it should be a subclass of `design`. Subclasses of `design` must obey the design protocol.

The fundamental operation of the CLIM graphic drawing model is to draw a design onto a drawing plane, thus drawing is always controlled by designs. The designs discussed in this chapter do the same thing at each point in the drawing plane. Chapter 14 discusses more general designs and reveals that regions are also designs.

A design can be characterized in several different ways:

All designs are either *bounded* or *unbounded*. Bounded designs are transparent everywhere beyond a certain distance from a certain point. Drawing a bounded design has no effect on the drawing plane outside that distance. Unbounded designs have points of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.

All designs are either *uniform* or *non-uniform*. Uniform designs have the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.

All designs are either *solid* or *translucent*. At each point a solid design is either completely opaque or completely transparent. A solid design can be opaque at some points and transparent at others. In translucent designs, at least one point has an opacity that is intermediate between completely opaque and transparent.

All designs are either *colorless* or *colored*. Drawing a colorless design uses a default color specified by the medium's foreground design. This is done by drawing with `(compose-in +foreground-ink+ the-colorless-design)`. See Chapter 14 for the details of `compose-in`.

⇒ `designp` *object* [*Function*]

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

13.3 Color

⇒ `color` [*Protocol Class*]

A member of the class `color` is a completely opaque design that represents the intuitive definition of color: white, black, red, pale yellow, and so forth. The visual appearance of a single point is completely described by its color. Drawing a color sets the color of every point in the drawing plane to that color, and sets the opacity to 1. The `color` class is the protocol class for a color, and is a subclass of `design`. If you want to create a new class that behaves like a color, it should be a subclass of `color`. Subclasses of `color` must obey the color protocol.

All of the standard instantiable color classes provided by CLIM are immutable.

A color can be specified by three real numbers between 0 and 1 (inclusive), giving the amounts of red, green, and blue. Three 0's mean black; three 1's mean white. The intensity-hue-saturation color model is also supported, but the red-green-blue color model is the primary model we will use in the specification.

⇒ `colorp` *object* [*Function*]

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

The following functions create colors. These functions produce objects that have equivalent effects; the only difference is in how the color components are specified. The resulting objects are indistinguishable when drawn. Whether these functions use the specified values exactly or approximate them because of limited color resolution is unspecified. Whether these functions create a new object or return an existing object with equivalent color component values is unspecified.

⇒ `make-rgb-color` *red green blue* [*Function*]

Returns a member of class `color`. The *red*, *green*, and *blue* arguments are real numbers between 0 and 1 (inclusive) that specify the values of the corresponding color components.

⇒ `make-ihs-color` *intensity hue saturation* [*Function*]

Returns a member of class `color`. The *intensity* argument is a real number between 0 and $\sqrt{3}$ (inclusive). The *hue* and *saturation* arguments are real numbers between 0 and 1 (inclusive).

⇒ `make-gray-color` *luminance* [*Function*]

Returns a member of class `color`. *luminance* is a real number between 0 and 1 (inclusive). On a black-on-white display device, 0 means black, 1 means white, and the values in between are shades of gray. On a white-on-black display device, 0 means white, 1 means black, and the values in between are shades of gray.

The following two functions comprise the color protocol. Both of them return the components of a color. All subclasses of `color` must implement methods for these generic functions.

⇒ `color-rgb` *color* [*Generic Function*]

Returns three values, the *red*, *green*, and *blue* components of the color *color*. The values are real numbers between 0 and 1 (inclusive).

⇒ `color-ihs` *color* [*Generic Function*]

Returns three values, the *intensity*, *hue*, and *saturation* components of the color *color*. The first value is a real number between 0 andn $\sqrt{3}$ (inclusive). The second and third values are real numbers between 0 and 1 (inclusive).

13.3.1 Standard Color Names

Table 13.1 lists all the provided color constants. Application programs can define other colors; these are provided because they are commonly used in the X Windows community, not because there is anything special about these particular colors. This table is a subset of the color listed in the file `/X11/R4/mit/rgb/rgb.txt`, from the X11 R4 distribution.

13.3.2 Contrasting Colors

⇒ `make-contrasting-inks` *n* `&optional` *k* [*Function*]

If *k* is not supplied, this returns a vector of *n* designs with recognizably different appearance. Elements of the vector are guaranteed to be acceptable values for the `:ink` argument to the drawing functions, and will not include `+foreground-ink+`, `+background-ink+`, or `nil`. Their class is otherwise unspecified. The vector is a fresh object that may be modified.

If *k* is supplied, it must be an integer between 0 and $n-1$ (inclusive), in which case `make-contrasting-inks` returns the *k*'th design rather than returning a vector of designs.

If the implementation does not have *n* different contrasting inks, `make-contrasting-inks` signals an error. This will not happen unless *n* is greater than eight.

+alice-blue+	+antique-white+	+aquamarine+
+azure+	+beige+	+bisque+
+black+	+blanched-almond+	+blue+
+blue-violet+	+brown+	+burlywood+
+cadet-blue+	+chartreuse+	+chocolate+
+coral+	+cornflower-blue+	+cornsilk+
+cyan+	+dark-goldenrod+	+dark-green+
+dark-khaki+	+dark-olive-green+	+dark-orange+
+dark-orchid+	+dark-salmon+	+dark-sea-green+
+dark-slate-blue+	+dark-slate-gray+	+dark-turquoise+
+dark-violet+	+deep-pink+	+deep-sky-blue+
+dim-gray+	+dodger-blue+	+firebrick+
+floral-white+	+forest-green+	+gainsboro+
+ghost-white+	+gold+	+goldenrod+
+gray+	+green+	+green-yellow+
+honeydew+	+hot-pink+	+indian-red+
+ivory+	+khaki+	+lavender+
+lavender-blush+	+lawn-green+	+lemon-chiffon+
+light-blue+	+light-coral+	+light-cyan+
+light-goldenrod+	+light-goldenrod-yellow+	+light-gray+
+light-pink+	+light-salmon+	+light-sea-green+
+light-sky-blue+	+light-slate-blue+	+light-slate-gray+
+light-steel-blue+	+light-yellow+	+lime-green+
+linen+	+magenta+	+maroon+
+medium-aquamarine+	+medium-blue+	+medium-orchid+
+medium-purple+	+medium-sea-green+	+medium-slate-blue+
+medium-spring-green+	+medium-turquoise+	+medium-violet-red+
+midnight-blue+	+mint-cream+	+misty-rose+
+moccasin+	+navajo-white+	+navy-blue+
+old-lace+	+olive-drab+	+orange+
+orange-red+	+orchid+	+pale-goldenrod+
+pale-green+	+pale-turquoise+	+pale-violet-red+
+papaya-whip+	+peach-puff+	+peru+
+pink+	+plum+	+powder-blue+
+purple+	+red+	+rosy-brown+
+royal-blue+	+saddle-brown+	+salmon+
+sandy-brown+	+sea-green+	+seashell+
+sienna+	+sky-blue+	+slate-blue+
+slate-gray+	+snow+	+spring-green+
+steel-blue+	+tan+	+thistle+
+tomato+	+turquoise+	+violet+
+violet-red+	+wheat+	+white+
+white-smoke+	+yellow+	+yellow-green+

Table 13.1: Table of Color Names

The rendering of the design may be a color or a stippled pattern, depending on whether the output medium supports color.

⇒ `contrasting-inks-limit` *port* [*Generic Function*]

Returns the number of contrasting colors (or stipple patterns if *port* is monochrome or grayscale) that can be rendered on any medium on the *port port*. Implementations are encouraged to make this as large as possible, but it must be at least 8. All classes that obey the medium protocol must implement a method for this generic function.

13.4 Opacity

⇒ `opacity` [*Protocol Class*]

A member of the class `opacity` is a completely colorless design that is typically used as the second argument to `compose-in` to adjust the opacity of another design. See Chapter 14 for the details of `compose-in`. The `opacity` class is the protocol class for an opacity, and is a subclass of `design`. If you want to create a new class that behaves like an opacity, it should be a subclass of `opacity`. Subclasses of `opacity` must obey the opacity protocol.

All of the standard instantiable opacity classes provided by CLIM are immutable.

Opacity controls how graphical output covers previous output. Opacity can vary from totally opaque to totally transparent. Intermediate opacity values result in color blending so that the earlier picture shows through what is drawn on top of it.

An opacity may be specified by a real number between 0 and 1 (inclusive). 0 is completely transparent, 1 is completely opaque, fractions are translucent. The opacity of a design is the degree to which it hides the previous contents of the drawing plane when it is drawn.

The fully transparent and fully opaque opacity levels (that is, opacities 0 and 1) must always be supported, but a valid CLIM implementation might only support a handful of opacity levels in between (including none). A valid CLIM implementation might implement color blending and unsaturated colors by stippling, although it is preferred, when possible, for a viewport to display a uniform color as a uniform color rather than as a perceptible stipple.

⇒ `opacityp` *object* [*Function*]

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

The following function returns an opacity:

⇒ `make-opacity` *value* [*Function*]

Returns a member of class `opacity` whose opacity is *value*, which is a real number in the range from 0 to 1 (inclusive), where 0 is fully transparent and 1 is fully opaque.

The following function returns the sole component of an opacity. This is the only function in the opacity protocol. All subclasses of `opacity` must implement methods for this generic function.

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

Returns the opacity value of the *opacity opacity*, which is a real number in the range from 0 to 1 (inclusive).

13.5 Color Blending

Drawing a design that is not completely opaque at all points allows the previous contents of the drawing plane to show through. The simplest case is drawing a solid design: where the design is opaque, it replaces the previous contents of the drawing plane; where the design is transparent, it leaves the drawing plane unchanged. In the more general case of drawing a translucent design, the resulting color is a blend of the design's color and the previous color of the drawing plane. For purposes of color blending, the drawn design is called the foreground and the drawing plane is called the background.

The function `compose-over` performs a similar operation: it combines two designs to produce a design, rather than combining a design and the contents of the drawing plane to produce the new contents of the drawing plane. For purposes of color blending, the first argument to `compose-over` is called the foreground and the second argument is called the background. See Chapter 14 for the details of `compose-over`.

Color blending is defined by an ideal function $\mathcal{F}:(r_1, g_1, b_1, o_1, r_2, g_2, b_2, o_2) \rightarrow (r_3, g_3, b_3, o_3)$ that operates on the color and opacity at a single point. (r_1, g_1, b_1, o_1) are the foreground color and opacity. (r_2, g_2, b_2, o_2) are the background color and opacity. (r_3, g_3, b_3, o_3) are the resulting color and opacity. The color blending function $\mathcal{F}$ is conceptually applied at every point in the drawing plane.

$\mathcal{F}$ performs linear interpolation on all four components:

$$
\begin{aligned}
o_3 &= \quad o_1 + (1 - o_1) * o_2 \\
r_3 &= (o_1 * r_1 + (1 - o_1) * o_2 * r_2)/o_3 \\
g_3 &= (o_1 * g_1 + (1 - o_1) * o_2 * g_2)/o_3 \\
b_3 &= (o_1 * b_1 + (1 - o_1) * o_2 * b_2)/o_3
\end{aligned}
$$

Note that if o_3 is zero, these equations would divide zero by zero. In that case r_3, g_3, and b_3 are defined to be zero.

CLIM requires that $\mathcal{F}$ be implemented exactly if o_1 is zero or one or if o_2 is zero. If o_1 is zero, the result is the background. If o_1 is one or o_2 is zero, the result is the foreground. For fractional opacity values, an implementation can deviate from the ideal color blending function either because the implementation has limited opacity resolution or because the implementation can compute a different color blending function much more quickly.

If a medium's background design is not completely opaque at all points, the consequences are unspecified. Consequently, a drawing plane is always opaque and drawing can use simplified color blending that assumes $o_2 = 1$ and $o_3 = 1$. However, `compose-over` must handle a non-opaque background correctly.

Note that these (r, g, b, o) quadruples of real numbers between 0 and 1 are mathematical and an implementation need not store information in this form. Most implementations are expected to use a different representation.

13.6 Indirect Inks

Drawing with an *indirect ink* is the same as drawing another design named directly. For example, `+foreground-ink+` is a design that draws the medium's foreground design and is the default value of the `:ink` drawing option. Indirect inks exist for the benefit of output recording. For example, one can draw with `+foreground-ink+`, change to a different `medium-foreground`, and replay the output record; the replayed output will come out in the new color.

You can change the foreground or background design of a medium at any time. This changes the contents of the medium's drawing plane. The effect is as if everything on the drawing plane is erased, the background design is drawn onto the drawing plane, and then everything that was ever drawn (provided it was saved in the output history) is drawn over again, using the medium's new foreground and background.

If an infinite recursion is created using an indirect ink, an error is signalled when the recursion is created, when the design is used for drawing, or both.

Two indirect inks are defined, but no advertised way is provided to create more of them.

⇒ `+foreground-ink+` [*Constant*]

An indirect ink that uses the medium's foreground design.

⇒ `+background-ink+` [*Constant*]

An indirect ink that uses the medium's background design.

13.7 Flipping Ink

⇒ `make-flipping-ink` *design1 design2* [*Function*]

Returns a design that interchanges occurrences of the two designs *design1* and *design2*. Drawing the resulting design over a background (either by drawing or with `compose-over`) is defined to change the color in the background that would have been drawn by *design1* at that point into the color that would have been drawn by *design2* at that point, and vice versa. The effect on any color other than the colors determined by those two designs is unspecified; however, drawing the same figure twice using the same flipping ink is guaranteed to be an "identity" operation. If either *design1* or *design2* is not solid, the consequences are unspecified. The purpose of flipping is to allow the use of "XOR hacks" for temporary changes to the display.

The opacity of a flipping ink is zero at points where the opacity of either *design1* or *design2* is zero. Otherwise the opacity of a flipping ink is 1. If *design1* or *design2* is translucent, the consequences are unspecified. If `compose-in` or `compose-out` is used to make a flipping ink translucent, the consequences are unspecified.

If *design1* and *design2* are equivalent, the result can be `+nowhere+`.

In Release 2, `make-flipping-ink` might require *design1* and *design2* to be colors.

⇒ `+flipping-ink+` [*Constant*]

A flipping ink that flips `+foreground-ink+` and `+background-ink+`.

13.8 Examples of Simple Drawing Effects

Drawing in the foreground color. Use the default, or specify `:ink +foreground-ink+`.

Erasing. Specify `:ink +background-ink+`.

Drawing in color. Specify `:ink +green+`, `:ink (make-rgb-color 0.6 0.0 0.4)`, and so forth.

Drawing an opaque gray. Specify `:ink (make-gray-color 0.25)` to draw in a shade of gray independent of the window's foreground color. On a non-color, non-grayscale display this will probably turn into a stipple.

Chapter 14

Patterned Designs

This chapter discusses more general designs than Chapter 13 and reveals that regions are also designs. This chapter generalizes to those designs that do not have the same color and opacity at every point in the drawing plane. These include:

- composite designs,
- patterns,
- stencils,
- tiled designs,
- transformed designs,
- output record designs, and
- regions

Several of the features described in this chapter may not be fully supported in Release 2 of CLIM.

Note: A design is a unification of both a shape and a color and opacity. As such, a design can serve multiple roles. For example, the same design can play the role of an ink that colors the drawing plane, a shape that specifies where to draw another design, a stencil that controls the compositing of two designs, the background of a window, or a region that defines clipping. It is important not to get confused between *type*, which is inherent in an object, and *role*, which is determined by how an object is used in a particular function call.

14.1 The Compositing Protocol

Compositing creates a design whose appearance at each point is a composite of the appearances of two other designs at that point. Three varieties of compositing are provided: *composing over*, *composing in*, and *composing out*.

The methods for `compose-over`, `compose-in`, and `compose-out` will typically specialize both of the design arguments.

In Release 2, compositing might only be supported for uniform designs.

⇒ `compose-over` *design1 design2* [*Generic Function*]

Composes a new design that is equivalent to the *design design1* on top of the *design design2*. Drawing the resulting design produces the same visual appearance as drawing *design2* and then drawing *design1*, but might be faster and might not allow the intermediate state to be visible on the screen.

If both arguments are regions, `compose-over` is the same as `region-union`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified. The result returned by `compose-over` might be freshly constructed or might be an existing object.

⇒ `compose-in` *ink mask* [*Generic Function*]

Composes a new design by clipping the *design ink* to the inside of the *design mask*. The first design, *ink*, supplies the color, while the second design, *mask*, changes the shape of the design by adjusting the opacity.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the same color that *ink* specifies. The opacity is the opacity that *ink* specifies, multiplied by the stencil opacity of *mask*.

The *stencil opacity* of a design at a point is defined as the opacity that would result from drawing the design onto a fictitious medium whose drawing plane is initially completely transparent black (opacity and all color components are zero), and whose foreground and background are both opaque black. With this definition, the stencil opacity of a member of class `opacity` is simply its value.

If *mask* is a solid design, the effect of `compose-in` is to clip *ink* to *mask*. If *mask* is translucent, the effect is a soft matte.

If both arguments are regions, `compose-in` is the same as `region-intersection`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified. The result returned by `compose-in` might be freshly constructed or might be an existing object.

⇒ `compose-out` *ink mask* [*Generic Function*]

Composes a new design by clipping the *design ink* to the outside of the *design mask*. The first design, *ink*, supplies the color, while the second design, *mask*, changes the shape of the design by adjusting the opacity.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the same color that *ink* specifies. The opacity is the opacity that *ink* specifies, multiplied by 1 minus the stencil opacity of *mask*.

If *mask* is a solid design, the effect of `compose-out` is to clip *ink* to the complement of *mask*. If *mask* is translucent, the effect is a soft matte.

If both arguments are regions, `compose-out` is the same as `region-difference` of *mask* and *ink*.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified. The result returned by `compose-out` might be freshly constructed or might be an existing object.

14.2 Patterns and Stencils

Patterning creates a bounded rectangular arrangement of designs, like a checkerboard. Drawing a pattern draws a different design in each rectangular cell of the pattern. To create an infinite pattern, apply `make-rectangular-tile` to a pattern.

A *stencil* is a special kind of pattern that contains only opacities.

⇒ `make-pattern` *array designs* [*Function*]

Returns a pattern design that has `(array-dimension` *array* `0)` cells in the vertical direction and `(array-dimension` *array* `1)` cells in the horizontal direction. *array* must be a two-dimensional array of non-negative integers less than the length of *designs*. *designs* must be a sequence of *designs*. The design in cell (i, j) of the resulting pattern is the nth element of *designs*, if n is the value of `(aref` *array* `i j)`. For example, *array* can be a bit-array and *designs* can be a list of two designs, the design drawn for 0 and the one drawn for 1.

Each cell of a pattern can be regarded as a hole that allows the design in it to show through. Each cell might have a different design in it. The portion of the design that shows through a hole is the portion on the part of the drawing plane where the hole is located. In other words, incorporating a design into a pattern does not change its alignment to the drawing plane, and does not apply a coordinate transformation to the design. Drawing a pattern collects the pieces of designs that show through all the holes and draws the pieces where the holes lie on the drawing plane. The pattern is completely transparent outside the area defined by the array.

Each cell of a pattern occupies a 1 by 1 square. You can use `transform-region` to scale the pattern to a different cell size and shape, or to rotate the pattern so that the rectangular cells become diamond-shaped. Applying a coordinate transformation to a pattern does not affect the designs that make up the pattern. It only changes the position, size, and shape of the cells' holes, allowing different portions of the designs in the cells to show through. Consequently, applying `make-rectangular-tile` to a pattern of nonuniform designs can produce a different appearance in each tile. The pattern cells' holes are tiled, but the designs in the cells are not tiled and a different portion of each of those designs shows through in each tile.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

⇒ `make-stencil` *array* [*Function*]

Returns a pattern design that has `(array-dimension` *array* `0)` cells in the vertical direction and `(array-dimension` *array* `1)` cells in the horizontal direction. *array* must be a two-dimensional array of real numbers between 0 and 1 (inclusive) that represent opacities. The design in cell (i, j) of the resulting pattern is the value of `(make-opacity (aref` *array* `i j))`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified.

14.3 Tiling

Tiling repeats a rectangular portion of a design throughout the drawing plane. This is most commonly used with patterns.

⇒ `make-rectangular-tile` *design width height* [*Function*]

Returns a design that, when used as an ink, tiles a rectangular portion of the *design design* across the entire drawing plane. The resulting design repeats with a period of *width* horizontally and *height* vertically. *width* and *height* must both be integers. The portion of *design* that appears in each tile is a rectangle whose top-left corner is at $(0, 0)$ and whose bottom-right corner is at $(width, height)$. The repetition of *design* is accomplished by applying a coordinate transformation to shift *design* into position for each tile, and then extracting a *width* by *height* portion of that design.

Applying a coordinate transformation to a rectangular tile does not change the portion of the argument *design* that appears in each tile. However, it can change the period, phase, and orientation of the repeated pattern of tiles. This is so that adjacent figures drawn using the same tile have their inks "line up".

14.4 Regions as Designs

Any member of the class `region` is a solid, colorless design. The design is opaque at points in the region and transparent elsewhere. Figure 14.1 shows how the design and region classes relate to each other.

Since bounded designs obey the region protocol, the functions `transform-region` and `untransform-region` accept any design as their second argument and apply a coordinate transformation to the design. The result is a design that might be freshly constructed or might be an existing object.

Transforming a uniform design simply returns the argument. Transforming a composite, flipping, or indirect design applies the transformation to the component design(s). Transforming a pattern, tile, or output record design is described in the sections on those designs.

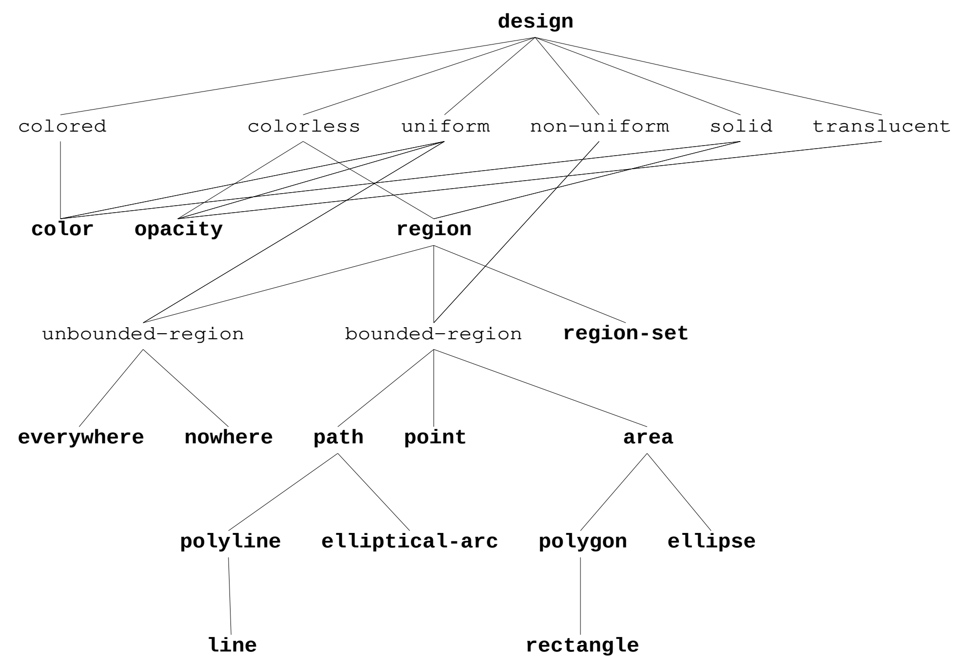


Figure 14.1: The class structure for all designs and regions. Entries in bold correspond to real CLIM classes.

14.5 Arbitrary Designs

⇒ `draw-design` *sheet design* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape text-style text-family text-face text-size* [*Function*]

Draws the design *design* onto the sheet *sheet*. This is defined to work for all types of regions and designs, although in practice some implementations may be more restrictive. The additional drawing arguments control the drawing of the design, depending on what sort of design is being drawn. For instance, if *design* is a path, then line style options may be supplied.

If *design* is an area, `draw-design` paints the specified region of the drawing plane with current ink of the sheet's medium. If *design* is a path, `draw-design` strokes the path with medium's current ink under control of the line-style. If *design* is a point, `draw-design` is the same as `draw-point`.

If *design* is a color or an opacity, `draw-design` paints the entire drawing plane (subject to the clipping region of the sheet's medium).

If *design* is `+nowhere+`, `draw-design` has no effect.

If *design* is a non-uniform design (see Chapter 14), `draw-design` paints the design, positioned at coordinates $(0,0)$.

CLIM implementations are required to support `draw-design` for the following cases:

- Designs created by the geometric object constructors, such as `make-line` and `make-ellipse`, in conjunction with drawing arguments that supply the drawing ink.
- Designs created by calling `compose-in` on a color and an object created by a geometric object constructor.

- Designs created by calling `compose-over` on any of the cases above.
- Designs returned by `make-design-from-output-record`.

⇒ `draw-pattern*` *sheet pattern x y* `&key` *clipping-region transformation* [*Function*]

Draws the pattern *pattern* on the sheet *sheet* at the position (x, y). *pattern* is any design created by `make-pattern`. *clipping-region* and *transformation* are as for `with-drawing-options` or any of the drawing functions.

Drawing a bitmap consists of drawing an appropriately aligned and scaled pattern constructed from the bitmap's bits. A 1 in the bitmap corresponds to `+foreground+`, while a 0 corresponds to `+background+` if an opaque drawing operation is desired, or to `+nowhere+` if a transparent drawing operation is desired.

Drawing a (colored) raster image consists of drawing an appropriately aligned and scaled pattern constructed from the raster array and raster color map.

`draw-pattern*` could be implemented as follows, assuming that the functions `pattern-width` and `pattern-height` return the width and height of the pattern.

```
(defun draw-pattern* (sheet pattern x y &key clipping-region transformation)
  (check-type pattern pattern)
  (let ((width (pattern-width pattern))
        (height (pattern-height pattern)))
    (if (or clipping-region transformation)
        (with-drawing-options (sheet :clipping-region clipping-region
                                     :transformation transformation)
          (draw-rectangle* sheet x y (+ x width) (+ y height)
                           :filled t :ink pattern))
        (draw-rectangle* sheet x y (+ x width) (+ y height)
                         :filled t :ink pattern))))
```

14.6 Examples of More Complex Drawing Effects

Painting a gray or colored wash over a display. Specify a translucent design as the ink, such as `:ink (compose-in +black+ (make-opacity 0.25))`, `:ink (compose-in +red+ (make-opacity 0.1))`, or `:ink (compose-in +foreground-ink+ (make-opacity 0.75))`. The last example can be abbreviated as `:ink (make-opacity 0.75)`. On a non-color, non-grayscale display this will probably turn into a stipple.

Drawing a faded but opaque version of the foreground color. Specify `:ink (compose-over (compose-in +foreground-ink+ (make-opacity 0.25)) +background-ink+)` to draw at 25% of the normal contrast. On a non-color, non-grayscale display this will probably turn into a stipple.

Drawing a tiled pattern. Specify `:ink (make-rectangular-tile (make-pattern` *array* *colors*`))`.

Drawing a “bitmap”. Use `(draw-design` *sheet* `(make-pattern` *bit-array* `(list +background-ink+ +foreground-ink+)) :transformation (make-translation-transformation` *x y*`))`.

14.7 Design Protocol

Issue: The generic functions underlying the functions described in this and the preceding chapter will be documented later. This will allow for programmer-defined design classes. — SWM

Part V

Extended Stream Output Facilities

Chapter 15

Extended Stream Output

CLIM provides a stream-oriented output layer that is implemented on top of the sheet output architecture. The basic CLIM output stream protocol is based on the character output 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.

15.1 Basic Output Streams

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

⇒ `standard-output-stream` [*Class*]

This class provides an implementation of the CLIM basic output stream protocol, based on the CLIM output kernel. Members of this class are mutable.

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

Writes the character *character* to the *output stream stream*, and returns *character* as its value.

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

This function returns the column number where the next character will be written on the *output stream stream*. The first column on a line is numbered 0.

⇒ `stream-start-line-p` *stream* [*Generic Function*]

Returns *true* if the *output stream stream* is positioned at the beginning of a line (that is, column 0), otherwise returns *false*.

⇒ `stream-write-string` *stream string* `&optional` *(start* `0`*) end* [*Generic Function*]

Writes the string *string* to the *output stream stream*. If *start* and *end* are supplied, they are integers that specify what part of *string* to output. *string* is returned as the value.

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

Writes an end of line character on the *output stream stream*, and returns *false*.

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

Writes an end of line character on the *output stream stream* only if the stream is not at the beginning of the line.

⇒ `stream-finish-output` *stream* [*Generic Function*]

Ensures that all the output sent to the *output stream stream* has reached its destination, and only then return *false*.

⇒ `stream-force-output` *stream* [*Generic Function*]

Like `stream-finish-output`, except that it may immediately return *false* without waiting for the output to complete.

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

Aborts any outstanding output operation in progress on the *output stream stream*, and returns *false*.

⇒ `stream-advance-to-column` *stream column* [*Generic Function*]

Writes enough blank space on the *output stream stream* so that the next character will be written at the position specified by *column*, which is an integer.

15.2 Extended Output Streams

In addition to the basic output stream protocol, CLIM defines an extended output stream protocol. This protocol extends the stream model to maintain the state of a text cursor, margins, text styles, inter-line spacing, and so forth.

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

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

⇒ `extended-output-stream-p` *object* [*Function*]

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

⇒ `:foreground` [*Init Arg*]

⇒ `:background` [*Init Arg*]

⇒ `:default-text-style` [*Init Arg*]

⇒ `:vertical-spacing` [*Init Arg*]

⇒ `:text-margin` [*Init Arg*]

⇒ `:end-of-line-action` [*Init Arg*]

⇒ `:end-of-page-action` [*Init Arg*]

⇒ `:default-view` [*Init Arg*]

All subclasses of `extended-output-stream` must handle these initargs, which are used to specify, respectively, the medium foreground and background, default text style, vertical spacing, default text margin, end of line and end of page actions, and the default view for the stream.

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

This class provides an implementation of the CLIM extended output stream protocol, based on the CLIM output kernel.

Members of this class are mutable.

15.3 The Text Cursor

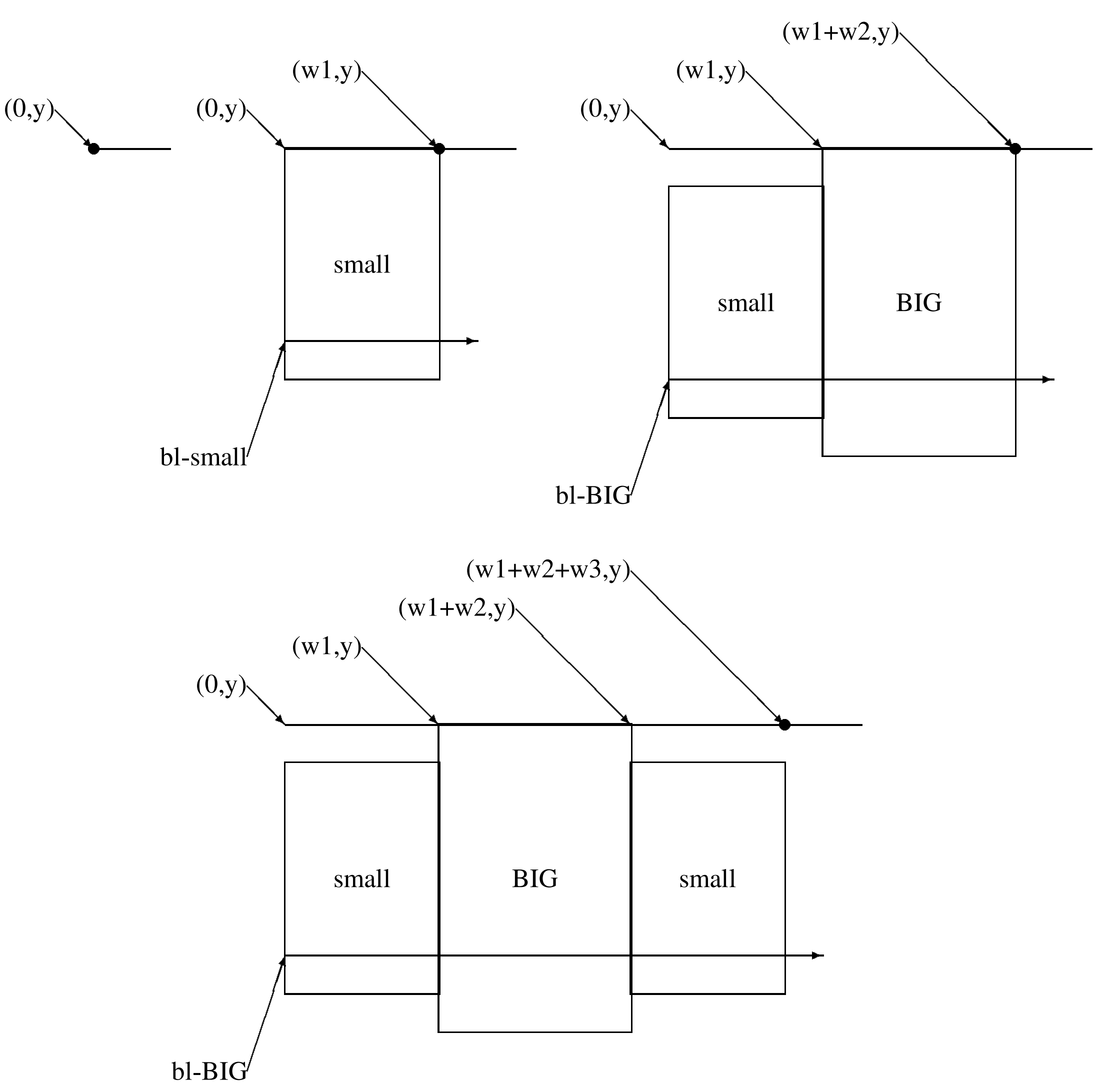
In the days when display devices displayed only two dimensional arrays of fixed width characters, the text cursor was a simple thing. A discrete position was selected in integer character units, and a character could go there and noplace else. Even for variable width fonts, simply addressing a character by the pixel position of one of its corners is sufficient. However, variable height fonts with variable baselines on pixel-addressable displays upset this simple model. The "logical" vertical reference point is the baseline, as it is in typesetting. In typesetting, however, an entire line of text is created with baselines aligned and padded to the maximum ascent and descent, then the entire line is put below the previous line.

It is clearly desirable to have the characters on a line aligned with their baselines, but when the line on the display is formed piece by piece, it is impossible to pick in advance the proper baseline. The solution CLIM adopts is to choose a baseline, but not commit to it.

The CLIM model says that text has at least 6 properties. With a reference point of $(0, 0)$ at the upper left of the text, it has a bounding box consisting of ascent, descent, left kerning, right extension, and a displacement to the next reference point in both x and y. CLIM determines the position of the reference point and draws the text relative to that, and then the cursor position is adjusted by the displacement. In this way, text has width and height, but the x and y displacements need not equal the width and height.

CLIM adopts the following approach to the actual rendering of a glyph. Textual output using the stream functions (*not* the graphics functions) maintains text on a "line". Note that a line is not an output record, but is rather a collection of "text so far", a top (which is positioned at the bottom of the previous line plus the stream's vertical spacing), a baseline, a bottom, and a "cursor position". The cursor position is defined to be at the top of the line, not at the baseline. The reason for this is that the baseline can move, but the top is relative to the previous line, which has been completed and therefore doesn't move. If text is drawn on the current line whose ascent is greater than the current ascent of the line, then the line is moved down to make room. This can be done easily using the output records for the existing text on the line. When there is enough room, the reference point for the text is the x position of the cursor at the baseline, and the cursor position is adjusted by the displacement.

The following figures show this in action before and after each of three characters are drawn. In all three figure, the small circle is the "cursor position". At first, there is nothing on the line. The first character establishes the initial baseline, and is then drawn. The upper left corner of the character is where the cursor was (as in the traditional model), but this will not remain the case. Drawing the second character, which is larger than the first, requires moving the first character down in order to get the baselines to align; during this time, the top of the line remains the same. Again, the upper left of the second character is where the cursor was, but that is no longer the case for the first character (which has moved down). The third character is smaller than the second, so no moving of characters needs to be done. However, the character is drawn to align the baselines, which in this case means the upper left is *not* where the cursor was. Nor is the cursor at the upper right of the character as it was for the previous two characters. It is, however, at the upper right of the collective line.



Issue: The above may be too simplistic. The displacement probably wants to depend not only on language but language rendering mode. For example, Japanese can apparently go either vertically or horizontally. It may be necessary to have the bounding box and perhaps the reference point dispatch as well. Similarly, "newline" could mean "down one line, all the way to the left" for English, "down one line, all the way to the right" for Arabic, or "to the left one line, all the way to the top." "Home cursor" is another ditty to consider. We need to discuss this on a larger scale with people who know multi-lingual rendering issues. — DCPL

15.3.1 Text Cursor Protocol

Many streams that maintain a text cursor display some visible indication of the text cursor. The object that represents this display is (somewhat confusingly) also called a cursor.

⇒ `cursor` [*Protocol Class*]

The protocol class that corresponds to cursors. If you want to create a new class that behaves like a cursor, it should be a subclass of `cursor`. Subclasses of `cursor` must obey the cursor protocol. Members of this class are mutable.

⇒ `cursorp` *object* [*Function*]

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

⇒ `:sheet` [*Init Arg*]

The `:sheet` initarg is used to specify the sheet with which the cursor is associated.

⇒ `standard-text-cursor` [*Class*]

The instantiable class that implements a text cursor. Typically, ports will further specialize this class.

⇒ `cursor-sheet` *cursor* [*Generic Function*]

Returns the sheet with which the *cursor cursor* is associated.

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

Returns the x and y position of the *cursor cursor* as two values.

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

Sets the x and y position of the *cursor cursor* to the specified position.

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

Returns (or sets) the visibility of the *cursor cursor*. The visibility can be either `:on` (meaning that the cursor will be visible at its current position), `:off` (meaning that the cursor is active, but not visible at its current position), or `nil` (meaning that the cursor is to be deactivated).

⇒ `display-cursor` *cursor state* [*Generic Function*]

This is responsible for drawing or erasing the *cursor cursor*. If *state* is `:draw`, the cursor will be drawn. Otherwise, if the *state* is `:erase`, the cursor will be erased.

15.3.2 Stream Text Cursor Protocol

The following generic functions comprise the stream text cursor protocol. Any extended output stream class must implement methods for these generic functions.

⇒ `stream-text-cursor` *stream* [*Generic Function*]

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

Returns (or sets) the text cursor object for the stream *stream*.

⇒ `stream-cursor-position*` *stream* [*Generic Function*]

Returns the current text cursor position for the *extended output stream stream* as two integer values, the x and y positions.

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

Sets the text cursor position of the *extended output stream stream* to *x* and *y*. *x* and *y* are in device units, and must be integers.

⇒ `stream-increment-cursor-position*` *stream dx dy* [*Generic Function*]

Moves the text cursor position of the *extended output stream stream* relatively, adding *dx* to the x coordinate and *dy* to the y coordinate. Either of *dx* or *dy* may be `nil`, meaning the the x or y cursor position will be unaffected. Otherwise, *dx* and *dy* must be integers.

15.4 Text Protocol

The following generic functions comprise the text protocol. Any extended output stream class must implement methods for these generic functions.

⇒ `stream-character-width` *stream character* `&key` *text-style* [*Generic Function*]

Returns a rational number corresponding to the amount of horizontal motion of the cursor position that would occur if the character *character* were output to the *extended output stream stream* in the *text style text-style* (which defaults to the current text style for the stream). This ignores the stream's text margin.

⇒ `stream-string-width` *stream character* `&key` *start end text-style* [*Generic Function*]

Computes how the cursor position would move horizontally if the string *string* were output to the *extended output stream stream* in the *text style text-style* (which defaults to the current text style for the stream) starting at the left margin. This ignores the stream's text margin.

The first returned value is the x coordinate that the cursor position would move to. The second returned value is the maximum x coordinate the cursor would visit during the output. (This is the same as the first value unless the string contains a `#\Newline` character.)

start and *end* are integers, and default to 0 and the length of the string, respectively.

⇒ `stream-text-margin` *stream* [*Generic Function*]
⇒ `(setf stream-text-margin)` *new-margin stream* [*Generic Function*]

The x coordinate at which text wraps around on the extended output stream *stream* (see `stream-end-of-line-action`). The default setting is the width of the viewport, which is the right-hand edge of the viewport when it is horizontally scrolled to the “initial position”.

You can use `setf` with `stream-text-margin` to establish a new text margin. If *new-margin* is `nil`, then the width of the viewport will be used. If the width of the viewport is later changed, the text margin will change, too.

⇒ `stream-line-height` *stream* `&key` *text-style* [*Generic Function*]

Returns what the line height of a line on the extended output stream *stream* containing text in the text style *text-style* would be, as a rational number. *text-style* defaults to the current text style for the stream.

⇒ `stream-vertical-spacing` *stream* [*Generic Function*]

Returns the current inter-line spacing (as a rational number) for the extended output stream *stream*.

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

Returns the current text baseline (as a rational number) for the extended output stream *stream*.

15.4.1 Mixing Text and Graphics

The following macro provides a convenient way to mix text and graphics on the same output stream.

⇒ `with-room-for-graphics` *(*`&optional` *stream* `&key` *(move-cursor* `t`*) height record-type)* `&body` *body* [*Macro*]

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

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

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

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

15.4.2 Wrapping of Text

⇒ `stream-end-of-line-action` *stream* [*Generic Function*]
⇒ `(setf stream-end-of-line-action)` *action stream* [*Generic Function*]

The end-of-line action controls what happens if the text cursor position moves horizontally out of the viewport, or if text output reaches the text margin. (By default the text margin is the width of the viewport, so these are usually the same thing.)

`stream-end-of-line-action` returns the end-of-line action for the *extended output stream stream*. It can be changed by using `setf` on `stream-end-of-line-action`.

The end-of-line action is one of:

- `:wrap`—when doing text output, wrap the text around (that is, break the text line and start another line). When setting the cursor position, scroll the window horizontally to keep the cursor position inside the viewport. This is the default.
- `:scroll`—scroll the window horizontally to keep the cursor position inside the viewport, then keep doing the output.
- `:allow`—ignore the text margin and do the output on the drawing plane beyond the visible part of the viewport.

⇒ `with-end-of-line-action` *(stream action)* `&body` *body* [*Macro*]

Temporarily changes *stream*'s end-of-line action for the duration of execution of body. *action* must be one of the actions described in `stream-end-of-line-action`.

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

⇒ `stream-end-of-page-action` *stream* [*Generic Function*]
⇒ `(setf stream-end-of-page-action)` *action stream* [*Generic Function*]

The end-of-page action controls what happens if the text cursor position moves vertically out of the viewport.

`stream-end-of-page-action` returns the end-of-page action for the *extended output stream stream*. It can be changed by using `setf` on `stream-end-of-page-action`.

The end-of-page action is one of:

- `:wrap`—when doing text output, wrap the text around (that is, go back to the top of the viewport).
- `:scroll`—scroll the window vertically to keep the cursor position inside the viewport, then keep doing output. This is the default.
- `:allow`—ignore the viewport and do the output on the drawing plane beyond the visible part of the viewport.

⇒ `with-end-of-page-action` *(stream action)* `&body` *body* [*Macro*]

Temporarily changes *stream*'s end-of-page action for the duration of execution of body. *action* must be one of the actions described in `stream-end-of-page-action`.

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

15.5 Attracting the User's Attention

⇒ `beep &optional` *sheet* [*Generic Function*]

Attracts the user's attention, usually with an audible sound.

15.6 Buffering of Output

Some sheets that support the output protocol may buffer output. When buffering is enabled on a sheet, the time at which output is actually done on the sheet is unpredictable. `force-output` or `finish-output` can be used to ensure that all pending output gets completed. If the sheet is a bidirectional stream, a `force-output` is performed whenever any sort of input is requested on the stream.

`with-buffered-output` provides a way to control when buffering is enabled on a sheet. By default, CLIM's interactive streams are buffered if the underlying window system supports buffering.

⇒ `sheet-buffering-output-p` *sheet* [*Generic Function*]

Returns *true* if the *sheet sheet* is currently buffering output, otherwise returns *false*.

⇒ `(setf sheet-buffering-output-p)` *buffer-p sheet* [*Generic Function*]

Sets `sheet-buffering-output-p` of the *sheet sheet* to *buffer-p*.

⇒ `with-output-buffered` *(sheet* `&optional` *(buffer-p* `t`*))* `&body` *body* [*Macro*]

If *buffer-p* is *true* (the default), this causes the *sheet* designated by *sheet* to start buffering output, and evaluates *body* in that context. If *buffer-p* is *false*, `force-output` will be called before *body* is evaluated. When *body* is exited (or aborted from), `force-output` will be called if output buffering will be disabled after `with-output-buffered` is exited.

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

Chapter 16

Output Recording

CLIM provides a mechanism whereby output (textual and graphical) may be captured into an *output history* for later replay on the same stream. This mechanism serves as the basis for many other tools, such as the formatted output and presentation mechanisms described elsewhere.

The output recording facility is layered on top of the graphics and text output protocols. It works by intercepting the operations in the graphics and text output protocols, and saving information about these operations in objects called *output records*. In general, an output record is a kind of display list, that is, a collection of instructions for drawing something on a stream. Some output records may have *children*, that is, a collection of inferior output records. Other output records, which are called *displayed output records*, correspond directly to displayed information on the stream, and do not have children.

If you think of output records being arranged in a tree, output records are the non-leaf nodes of the tree, for example, the top-level output history of an output recording stream. Displayed output records are all of the leaf nodes in the tree, for example, displayed text and graphics records.

Displayed output records must record the state of the supplied drawing options at the instant the output record is created, as follows. The ink supplied by the programmer must be captured without resolving indirect inks; this is so that a user can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay. The specified "user" transformation (composed with the medium transformation), clipping region, and line style must be captured in the output record; subsequent replaying of the record under a new user transformation, clipping region, or line style will not affect the replayed output. CLIM implementation are permitted to capture the text style either fully merged against the medium's default, or not; in the former case, subsequent changes to the medium's default text style will not affect replaying the record, but in the latter case changing the default text style will affect replaying.

A CLIM stream that supports output recording has an output history object, which is a special kind of output record that supports some other operations. CLIM defines a standard set of output history implementations and a standard set of output record types.

The output recording mechanism is defined so as to permit application specific or host window system specific implementations of the various recording protocols. CLIM implementations should provide several types of standard output records with different characteristics for search, storage, and retrieval. Two examples are “sequence” output records (which store elements in a sequence, and whose insertion and retrieval complexity is O(n)) and “tree” output records (which store elements in some sort of tree based on the location of the element, and whose insertion and retrieval complexity is O(n log n)).

16.1 Output Records

⇒ `output-record` [*Protocol Class*]

The protocol class that is used to indicate that an object is an output record, that is, an object that contains other output records. This is a subclass of `bounding-rectangle`, and as such, output records obey the bounding rectangle protocol. If you want to create a new class that behaves like an output record, it should be a subclass of `output-record`. Subclasses of `output-record` must obey the output record protocol.

All output records are mutable.

⇒ `output-record-p` *object* [*Function*]

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

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

The protocol class that is used to indicate that an object is a displayed output record, that is, an object that represents a visible piece of output on some output stream. This is a subclass of `bounding-rectangle`. If you want to create a new class that behaves like a displayed output record, it should be a subclass of `displayed-output-record`. Subclasses of `displayed-output-record` must obey the displayed output record protocol.

All displayed output records are mutable.

⇒ `displayed-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *displayed output record*, otherwise returns *false*.

⇒ `:x-position` [*Init Arg*]

⇒ `:y-position` [*Init Arg*]

⇒ `:parent` [*Init Arg*]

All subclasses of either `output-record` or `displayed-output-record` must handle these three initargs, which are used to specify, respectively, the x and y position of the output record, and the parent of the output record.

⇒ `:size` [*Init Arg*]

All subclasses of `output-record` must handle the `:size` initarg. It is used to specify how much room should be left for child output records (if, for example, the children are stored in a vector). It is permissible for `:size` to be ignored, provided that the resulting output record is able to store the specified number of child output records.

16.1.1 The Basic Output Record Protocol

All subclasses of `output-record` and `displayed-output-record` must inherit or implement methods for the following generic functions.

When the generic functions in this section take both *record* and a *stream* arguments, CLIM implementations will specialize the *stream* argument for the `standard-output-recording-stream` class and the *record* argument for all of the implementation-specific output record classes.

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

Returns the x and y position of the *output record record* as two rational numbers. The position of an output record is the position of the upper-left corner of its bounding rectangle. The position is relative to the stream, where $(0, 0)$ is (initially) the upper-left corner of the stream.

⇒ `(setf output-record-position*)` *new-x new-y record* [*Generic Function*]

Changes the x and y position of the *output record record* to be *new-x* and *new-y* (which are rational numbers), and updates the bounding rectangle to reflect the new position (and saved cursor positions, if the output record stores it). If *record* has any children, all of the children (and their descendants as well) will be moved by the same amount as *record* was moved. The bounding rectangles of all of *record*'s ancestors will also be updated to be large enough to contain *record*. This does not replay the output record, but the next time the output record is replayed it will appear at the new position.

⇒ `output-record-start-cursor-position*` *record* [*Generic Function*]

Returns the x and y starting cursor position of the *output record record* as two integer values. The positions are relative to the stream, where $(0, 0)$ is (initially) the upper-left corner of the stream.

Text output records and updating output records maintain the cursor position. Graphical output records and other output records that do not require or affect the cursor position will return `nil` as both of the values.

⇒ `(setf output-record-start-cursor-position*)` *new-x new-y record* [*Generic Function*]

Changes the x and y starting cursor position of the *output record record* to be *new-x* and *new-y* (which are integers). This does not affect the bounding rectangle of *record*, nor does it replay the output record. For those output records that do not require or affect the cursor position, the method for this function is a no-op.

⇒ `output-record-end-cursor-position*` *record* [*Generic Function*]

Returns the x and y ending cursor position of the *output record record* as two integer values. The positions are relative to the stream, where $(0,0)$ is (initially) the upper-left corner of the stream. Graphical output records do not track the cursor position, so only text output record (and some others) will return meaningful values for this.

Text output records and updating output records maintain the cursor position. Graphical output records and other output records that do not require or affect the cursor position will return `nil` as both of the values.

⇒ `(setf output-record-end-cursor-position*)` *new-x new-y record* [*Generic Function*]

Changes the x and y ending cursor position of the output record *record* to be *new-x* and *new-y* (which are integers). This does not affect the bounding rectangle of *record*, nor does it replay the output record. For those output records that do not require or affect the cursor position, the method for this function is a no-op.

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

Returns the output record that is the parent of the output record *record*, or `nil` if the record has no parent.

⇒ `replay` *record stream* `&optional` *region* [*Function*]

This function bind `stream-recording-p` of *stream* to false, and then calls `replay-output-record` on the arguments *record*, *stream*, and *region*. If `stream-drawing-p` of *stream* is false, `replay` does nothing. `replay` is typically called during scrolling, by repaint handlers, and so on.

CLIM implementations are permitted to default *region* either to `nil` or to the region corresponding to viewport of *stream*.

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

Displays the output captured by the output record *record* on the output recording stream *stream*, exactly as it was originally captured (subject to subsequent modifications). The current user transformation, line style, text style, ink, and clipping region of *stream* are all ignored during the replay operation. Instead, these are gotten from the output record.

If *record* is not a displayed output record, then replaying it involves replaying all of its children. If *record* is a displayed output record, then replaying it involves redoing the graphics operation captured in the record.

region is a region that can be supplied to limit what records are displayed. Only those records that overlap *region* are replayed. The default for *region* is `+everywhere+`.

It is permissible for implementations to restrict `replay-output-record` such that *stream* must be the same stream on which the output records were originally recorded.

Issue: How does replaying a text output record (or any record that maintains the cursor position) affect the cursor position of the stream? — SWM

⇒ `erase-output-record` *record stream* [*Generic Function*]

Erases the *output record record* from the *output recording stream stream*, removes *record* from *stream*'s output history, and ensures that all other output records that were covered by *record* are visible. In effect, this draws background ink over the record, and then redraws all the records that overlap *record*.

⇒ `output-record-hit-detection-rectangle*` *record* [*Generic Function*]

This method is used to establish the usual "effective size" of *record* for hit detection queries. Four values are returned corresponding to the edges of the bounding rectangle that is the hit detection rectangle. The default method (on CLIM's standard output record class) is equivalent to calling calling `bounding-rectangle*` on *record*, but this method can be specialized to return a larger bounding rectangle in order to implement a form of hysteresis.

⇒ `output-record-refined-sensitivity-test` *record x y* [*Generic Function*]

This is used to definitively answer hit detection queries, such as finding the output record that is located at the coordinates of a pointing device. Once the position (x, y) has been determined to lie within `output-record-hit-detection-rectangle*`,`output-record-refined-sensitivity-test` is invoked. Output record subclasses can provide a method that provides a more precise definition of a hit, for example, output records for elliptical rings will implement a method that detects whether the pointing device is on the elliptical ring.

⇒ `highlight-output-record` *record stream state* [*Generic Function*]

This method is called in order to draw a highlighting box around the *output record record* on the *output recording stream stream*. *state* will be either `:highlight` (meaning to draw the highlighting) or `:unhighlight` (meaning to erase the highlighting). The default method (on CLIM's standard output record class) will simply draw a rectangle that corresponds the the bounding rectangle of *record*.

16.1.2 The Output Record "Database" Protocol

All classes that are subclasses of `output-record` must implement methods for the following generic functions.

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

Returns a sequence of all of the children of the *output record record*. It is unspecified whether the sequence is a freshly created object or a "live" object representing the state of *record*.

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

Adds the new *output record child* to the *output record record*. The bounding rectangle for *record* (and all its ancestors) must be updated to account for the new child record.

The methods for the `add-output-record` will typically specialize only the *record* argument.

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

Removes the *output record child* from the *output record record*. The bounding rectangle for *record* (and all its ancestors) may be updated to account for the child having been removed, although this is not mandatory.

If *errorp* is true (the default) and *child* is not contained within *record*, then an error is signalled.

The methods for the `delete-output-record` will typically specialize only the *record* argument.

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

Removes all of the children from the *output record record*, and resets the bounding rectangle of *record* to its initial state.

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

Returns the number of children contained within the *output record record*.

⇒ `map-over-output-records-containing-point*` *function record x y* [*Generic Function*]

Maps over all of the children of the *output record record* that contain the point at (x, y), calling *function* on each one. *function* is a function of one argument, the record containing the point; it has dynamic extent.

If there are multiple records that contain the point and that overlap each other, `map-over-output-records-containing-point*` must hit the uppermost (most recently inserted) record first and the bottommost (least recently inserted) record last. Otherwise, the order in which the records are traversed is unspecified.

⇒ `map-over-output-records-overlapping-region` *function record region* [*Generic Function*]

Maps over all of the children of the *output record record* that overlap the *region region*, calling *function* on each one. *function* is a function of one argument, the record overlapping the region; it has dynamic extent.

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

16.1.3 Output Record Change Notification Protocol

The following functions are called by programmers and by CLIM itself in order to notify a parent output record when the bounding rectangle of one of its child output record changes.

⇒ `recompute-extent-for-new-child` *record child* [*Generic Function*]

This function is called whenever a new child is added to an output record. Its contract is to update the bounding rectangle of the *output record record* to be large enough to completely contain the new child *output record child*. The parent of *record* must be notified by calling `recompute-extent-for-`

`changed-child`.

`add-output-record` is required to call `recompute-extent-for-new-child`.

⇒ `recompute-extent-for-changed-child` *record child old-min-x old-min-y old-max-x old-max-y* [*Generic Function*]

This function is called whenever the bounding rectangle of one of the childs of a record has been changed. Its contract is to update the bounding rectangle of the *output record record* to be large enough to completely contain the new bounding rectangle of the child *output record child*. All of the ancestors of *record* must be notified by recursively calling `recompute-extent-for-changed-child`.

Whenever the bounding rectangle of an output record is changed or `delete-output-record` is called, `recompute-extent-for-changed-child` must be called to inform the parent of the record that a change has taken place.

⇒ `tree-recompute-extent` *record* [*Generic Function*]

This function is called whenever the bounding rectangles of a number of children of a record have been changed, such as happens during table and graph formatting. Its contract is to compute the bounding rectangle large enough to contain all of the children of the output record *record*, adjust the bounding rectangle of the *output record record* accordingly, and then call `recompute-extent-for-changed-child` on *record*.

16.2 Types of Output Records

This section discusses several types of output records, including two standard classes of output records and the displayed output record protocol.

16.2.1 Standard Output Record Classes

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

The standard class provided by CLIM to store a relatively short sequence of output records; a subclass of `output-record`. The insertion and retrieval complexity of this class is O(n). Most of the formatted output facilities (such as `formatting-table`) create output records that are a subclass of `standard-sequence-output-record`.

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

The standard class provided by CLIM to store longer sequences of output records. Typically, the child records of a tree output record will be maintained in some sort of sorted order, such as a lexicographic ordering on the x and y coordinates of the children. The insertion and retrieval complexity of this class is O(n log n).

16.2.2 Graphics Displayed Output Records

Graphics displayed output records are used to record the output produced by the graphics functions, such as `draw-line*`. Each graphics displayed output record describes the output produced by a call to one of the graphics functions.

The exact contents of graphics displayed output records is unspecified, but they must store sufficient information to be able to exactly redraw the original output at replay time. The minimum information that must be captured for all graphics displayed output records is as follows:

- The description of the graphical object itself, for example, the end points of a line or the center point and radius of a circle.
- The programmer-supplied ink at the time the drawing function was called. Indirect inks must not be resolved, so that a user can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay.
- For paths, the programmer-supplied line-style at the time the drawing function was called.
- The programmer-supplied clipping region at the time the drawing function was called.
- The medium's user transformation. This may be accomplished by either explicitly storing the transformation, or by transforming the coordinates supplied to the drawing function and capturing the transformed coordinates.

⇒ `graphics-displayed-output-record` [*Protocol Class*]

The protocol class that corresponds to output records for the graphics functions, such as `draw-line*`. This is a subclass of `displayed-output-record`. If you want to create a new class that behaves like a graphics displayed output record, it should be a subclass of `graphics-displayed-output-record`. Subclasses of `graphics-displayed-output-record` must obey the graphics displayed output record protocol.

⇒ `graphics-displayed-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *graphics displayed output record*, otherwise returns *false*.

16.2.3 Text Displayed Output Record

Text displayed output records are used to record the textual output produced by such functions as `stream-write-char` and `stream-write-string`. Each text displayed output record corresponds to no more than one line of textual output (that is, line breaks caused by `terpri` and `fresh-line` create a new text output record, as do certain other stream operations described below).

The exact contents of text displayed output records is unspecified, but they must store sufficient information to be able to exactly redraw the original output at replay time. The minimum information that must be captured for all text displayed output records is as follows:

- The displayed text strings.
- The starting and ending cursor positions.
- The text style in which the text string was written. Depending on the CLIM implementation, this may be either fully merged against the medium's default or not; in the former case, subsequent changes to the medium's default text style will not affect replaying the record, but in the latter case changing the default text style will affect replaying.
- The programmer-supplied ink at the time the drawing function was called. Indirect inks must not be resolved, so that a user can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay.
- The programmer-supplied clipping region at the time the drawing function was called.

⇒ `text-displayed-output-record` [*Protocol Class*]

The protocol class that corresponds to text displayed output records. This is a subclass of `displayed-output-record`. If you want to create a new class that behaves like a text displayed output record, it should be a subclass of `text-displayed-output-record`. Subclasses of `text-displayed-output-record` must obey the text displayed output record protocol.

⇒ `text-displayed-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *text displayed output record*, otherwise returns *false*.

The following two generic functions comprise the text displayed output record protocol.

⇒ `add-character-output-to-text-record` *text-record character text-style width height baseline* [*Generic Function*]

Adds the character *character* to the *text displayed output record* *text-record* in the text style *text-style*. *width* and *height* are the width and height of the character in device units, and are used to compute the bounding rectangle for the text record. *baseline* is the new baseline for characters in the output record.

⇒ `add-string-output-to-text-record` *text-record string start end text-style width height baseline* [*Generic Function*]

Adds the string *string* to the *text displayed output record* *text-record* in the text style *text-style*. *start* and *end* are integers that specify the substring within *string* to add to the text output record. *width* and *height* are the width and height of the character in device units, and are used to compute the bounding rectangle for the text record. *baseline* is the new baseline for characters in the output record.

⇒ `text-displayed-output-record-string` *text-record* [*Generic Function*]

Returns the string contained by the *text displayed output record* *text-record*. This function returns objects that reveal CLIM's internal state; do not modify those objects.

16.2.4 Top-Level Output Records

Top-level output records are similar to ordinary output records, except that they must maintain additional state, such as the information required to display scroll bars.

⇒ `stream-output-history-mixin` [*Class*]

This class is mixed into some other output record class to produce a new class that is suitable for use as a a top-level output history.

When the bounding rectangle of an member of this class is updated, CLIM implementations must update any window decorations (such as scroll bars) associated with the stream with which the *output record history* is associated.

16.3 Output Recording Streams

CLIM defines an extension to the stream protocol that supports output recording. The stream has an associated output history record and provides controls to enable and disable output recording.

Issue: Do we want to support only output recording streams, or do we want to support output recording sheets as well? If the latter, we need to split apart graphics output recording and textual output recording, and rename lots of things. — SWM

⇒ `output-recording-stream` [*Protocol Class*]

The protocol class that indicates that a stream is an output recording stream. If you want to create a new class that behaves like an output recording stream, it should be a subclass of `output-recording-stream`. Subclasses of `output-recording-stream` must obey the output recording stream protocol.

⇒ `output-recording-stream-p` *object* [*Function*]

Returns *true* if *object* is an *output recording stream*, otherwise returns *false*.

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

The class used by CLIM to implement output record streams. This is a subclass of `output-recording-stream`. Members of this class are mutable.

16.3.1 The Output Recording Stream Protocol

The following generic functions comprise the output recording stream protocol. All subclasses of `output-recording-stream` must implement methods for these generic functions.

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

Returns *true* when the *output recording stream stream* is recording all output performed to it, otherwise returns *false*.

⇒ `(setf stream-recording-p)` *value stream* [*Generic Function*]

Changes the state of `stream-recording-p` to be *value*, which must be either `t` or `nil`.

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

Returns *true* when the *output recording stream stream* will actually draw on the viewport when output is being performed to it, otherwise returns *false*.

⇒ `(setf stream-recording-p)` *value stream* [*Generic Function*]

Changes the state of `stream-recording-p` to be *value*, which must be either `t` or `nil`.

⇒ `stream-output-history` *stream* [*Generic Function*]

Returns the history (or top-level output record) for the *output recording stream stream*.

⇒ `stream-current-output-record` *stream* [*Generic Function*]

The current "open" output record for the *output recording stream stream*, the one to which `stream-add-output-record` will add a new child record. Initially, this is the same as `stream-output-history`. As nested output records are created, this acts as a "stack".

⇒ `(setf stream-current-output-record)` *record stream* [*Generic Function*]

Sets the current "open" output record for the *output recording stream stream* to the *output record record*.

⇒ `stream-add-output-record` *stream record* [*Generic Function*]

Adds the *output record record* to the current output record on the *output recording stream stream* (that is, `stream-current-output-record`).

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

Directs the *output recording stream stream* stream to invoke `replay` on its output history. Only those records that overlap the *region region* (which defaults to the viewport of the stream) are replayed.

16.3.2 Graphics Output Recording

Using `draw-line*` as an example, calling any of the drawing functions specified in Section 12.4 and Section 12.6 results in the following series of function calls on an output recording stream:

- A program calls `draw-line*` on arguments *sheet*, *x1*, *y1*, *x2*, and *y2*, and perhaps some drawing options.

- `draw-line*` merges the supplied drawing options into the sheet's medium, and then calls `medium-draw-line` on the sheet. (Note that a compiler macro could detect the case where there are no drawing options or constant drawing options, and do this at compile time.)
- The `:around` method for `medium-draw-line` on the output recording stream is called. This creates an output record with all of the information necessary to replay the output record, if `stream-recording-p` is *true*. If `stream-drawing-p` is *true*, this then does a `call-next-method`. (Note that the `call-next-method` could be replaced by a call to the `medium-draw-line` method on the sheet's medium, avoiding the need for a trampolining function call.)
- `medium-draw-line` performs the necessary user transformations by applying the medium transformation to *x1*, *y1*, *x2*, and *y2*, and to the clipping region. Then `medium-draw-line` calls `port-draw-line` to do the actual drawing.

`replay-output-record` for a graphics displayed output record simply calls the medium drawing function (such as `medium-draw-line`) directly on the sheet (*not* on the medium).

16.3.3 Text Output Recording

Issue: This is the place where `write-string` *and friends get captured in order to create output record. The generic functions include things like* `stream-write-string`*, which are specialized by output recording streams to do the output recording. Describe exactly what happens. — SWM*

⇒ `stream-text-output-record` *stream text-style* [*Generic Function*]

Returns a text output record for the *output recording stream stream* suitable for holding characters in the text style *text-style*. If there is a currently "open" text output record that can hold characters in the specified text style, it is simply returned. Otherwise a new text output record is created that can hold characters in that text style, and its starting cursor position set to the cursor position of *stream*.

⇒ `stream-close-text-output-record` *stream* [*Generic Function*]

Closes the *output recording stream stream*'s currently "open" text output record by recording the stream's current cursor position as the ending cursor position of the record and adding the text output record to *stream*'s current output record by calling `stream-add-output-record`.

If there is no "open" text output record, `stream-close-text-output-record` does nothing.

Calling `stream-finish-output` or `stream-force-output`, calling `redisplay`, setting the text cursor position (via `stream-set-cursor-position*`, `terpri`, or `fresh-line`), creating a new output record (for example, via `with-new-output-record`), or changing the state of `stream-recording-p` must close the current text output record. Some CLIM implementations may also choose to close the current text output record when the stream's drawing options or text style are changed, depending on the exact implementation of text output records.

⇒ `stream-add-character-output` *stream character text-style width height baseline* [*Generic Function*]

Adds the character *character* to the output recording stream *stream*'s text output record in the text style *text-style*. *width* and *height* are the width and height of the character in device units. *baseline* is the new baseline for the stream. `stream-add-character-output` must be implemented by calling `add-character-output-to-text-record`.

`stream-write-char` on an output recording stream will call `stream-add-character-output` when `stream-recording-p` is *true*.

⇒ `stream-add-string-output` *stream string start end text-style width height baseline* [*Generic Function*]

Adds the string *string* to the output recording stream *stream*'s text output record in the text style *text-style*. *start* and *end* are integers that specify the substring within *string* to add to the text output record. *width* and *height* are the width and height of the string in device units. *baseline* is the new baseline for the stream. `stream-add-string-output` must be implemented by calling `add-string-output-to-text-record`.

`stream-write-string` on an output recording stream will call `stream-add-string-output` when `stream-recording-p` is *true*.

16.3.4 Output Recording Utilities

CLIM provides several helper macros to control the output recording facility.

⇒ `with-output-recording-options` *(stream* `&key` *record draw)* `&body` *body* [*Macro*]

Enables or disables output recording and/or drawing on the output recording stream designated by *stream*, within the extent of *body*.

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.

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

⇒ `invoke-with-output-recording-options` *stream function record draw* [*Generic Function*]

Enables or disables output recording and/or drawing on the output recording stream *stream*, and calls the function *function* with the new output recording options in effect. *function* is a function of one argument, the stream; it has dynamic extent.

If *draw* is *false*, output to the stream is not drawn on the viewport, but recording proceeds according to *record*; if *draw* is *true*, the output is drawn. If *record* is `nil`, output recording is disabled, but output otherwise proceeds according to *draw*; if *draw* is *true*, output recording is enabled.

All output recording streams must implement a method for `invoke-with-output-recording-options`.

⇒ `with-new-output-record` *(stream* `&optional` *record-type record* `&rest` *init-args)* `&body` *body* [*Macro*]

Creates a new output record of type *record-type* (which defaults to `standard-sequence-output-record`) and then captures the output of *body* into the new output record, and inserts the new record into the current "open" output record associated with the *output recording stream* designated by *stream*. While *body* is being evaluated, the current output record for *stream* will be bound to the new output record.

If *record* is supplied, it is the name of a variable that will be lexically bound to the new output record inside of body. *init-args* are CLOS initialization arguments that are passed to `make-instance` when the new output record is created.

`with-new-output-record` returns the output record it creates.

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.

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

⇒ `invoke-with-new-output-record` *stream function record-type* `&rest` *init-args* `&key` [*Generic Function*]

Creates a new output record of type *record-type*. The function *function* is then called, and any output it does to the *output recording stream stream* is captured in the new output record. The new record is then inserted into the current "open" output record associated with *stream* (or the top-level output record if there is no currently "open" one). While *function* is being executed, the current output record for *stream* will be bound to the new output record.

function is a function of two arguments, the stream and the output record; it has dynamic extent. *init-args* are CLOS initialization arguments that are passed to `make-instance` when the new output record is created.

`invoke-with-new-output-record` returns the output record it creates.

All output recording streams must implement a method for `invoke-with-new-output-record`.

⇒ `with-output-to-output-record` *(stream* `&optional` *record-type record* `&rest` *init-args))* `&body` *body* [*Macro*]

This is identical to `with-new-output-record` except that the new output record is not inserted into the output record hierarchy, and the text cursor position of *stream* is initially bound to $(0,0)$.

record-type is the type of output record to create, which defaults to `standard-sequence-output-`

`record`. *init-args* are CLOS initialization arguments that are passed to `make-instance` when the new output record is created.

If *record* is supplied, it is a variable that will be bound to the new output record while body is evaluated.

`with-output-to-output-record` returns the output record it creates.

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. Unlike facilities such as `with-output-to-string`, *stream* must be an actual stream, but no output will be done to it. *body* may have zero or more declarations as its first forms.

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

⇒ `invoke-with-output-to-output-record` *stream function record-type* `&rest` *init-args* `&key` [*Generic Function*]

This is similar to `invoke-with-new-output-record` except that the new output record is not inserted into the output record hierarchy, and the text cursor position of *stream* is initially bound to $(0, 0)$. That is, when `invoke-with-output-to-output-record` is used, no drawing on the stream occurs and nothing is put into the stream's normal output history. The function *function* is called, and any output it does to *stream* is captured in the output record.

function is a function of two arguments, the stream and the output record; it has dynamic extent. *record-type* is the type of output record to create. *init-args* are CLOS initialization arguments that are passed to `make-instance` when the new output record is created.

`invoke-with-output-to-output-record` returns the output record it creates.

All output recording streams must implement a method for `invoke-with-output-to-output-record`.

⇒ `make-design-from-output-record` *record* [*Generic Function*]

Makes a design that replays the output record *record* when drawn. If *record* is changed after the design is made, the consequences are unspecified. Applying a coordinate transformation to the design changes the transformation used when replaying the output record.

It is permissible for implementations to support this only for those output records that correspond to the geometric object classes (for example, the output records created by `draw-line*` and `draw-ellipse*`).

Chapter 17

Table Formatting

CLIM provides a mechanism for tabular formatting of arbitrary output.

To employ these facilities the programmer annotates some output-generating code with advisory macros that describe the high-level formatting constraints, for example, what parts of code produce a row of the table, what parts of that produce the cells in the row.

For example, the following produces a table consisting of three columns containing a number, its square, and its cube. The output can be seen in Figure 17.1.

```
(defun table-test (count stream)
  (fresh-line stream)
  (formatting-table (stream :x-spacing '(3 :character))
    (dotimes (i count)
      (formatting-row (stream)
        (formatting-cell (stream :align-x :right)
          (prin1 i stream))
        (formatting-cell (stream :align-x :right)
          (prin1 (* i i) stream))
        (formatting-cell (stream :align-x :right)
          (prin1 (* i i i) stream))))))
```

The general contract of these facilities is described in the next section.

17.1 Overview of Table Formatting Facilities

In general, table formatting involves a sharing of responsibilities between user-written code and CLIM code. Code that employs only the lower level output facilities has full control over "where every piece of ink goes" in the output. In contrast, code that employs CLIM's table formatting facilities passes control

0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729

Figure 17.1: Example of tabular output.

to CLIM at a higher level. The programmer benefits by being able to specify the appearance of output in more compact abstract terms, and by not having to write the code that constrains the output to appear in proper tabular form.

Tabular output consists of a rectangular array of pieces of output corresponding to the bounding rectangles of the output. Each piece of output forms the contents of a *table cell*. There is no restriction on the contents of a table cell; cells may contain text, graphics, even other tables. For purposes of this discussion, we draw a strong distinction between specifying what goes in a cell, and specifying how the cells are arranged to form a table.

Specifying the contents of a cell is the responsibility of the programmer. A programmer using the table formatting facilities can predict the appearance of any individual cell by simply looking at the code for that cell. A cell's appearance does not depend upon where in the table it lies, for instance. The only thing about a cell's appearance that cannot be predicted from that cell alone is the amount of space the table formatting has to introduce in order to perform the desired alignment.

Specifying the relative arrangements of cells to form a table is the responsibility of CLIM based on the advice of the programmer. The programmer advises CLIM about extra space to put between rows or columns, for instance, but does not directly control the absolute positioning of a cell's contents.

For purposes of understanding table formatting, the following model may be used.

- The code for a cell draws to a stream that has a "private" (local to that cell) drawing plane.
- After output for a cell has finished, the bounding rectangle of all output on the "private" drawing plane is found. The region within that bounding rectangle forms the contents of a cell.
- Additional rectangular regions, containing only background ink, are attached to the edges of the cell contents. These regions ensure that the cells satisfy the tabular constraints that within a row all cells have the same height, and within a column all cells have the same width. CLIM may also introduce additional background for other purposes as described below.
- The cells are assembled into rows and columns.

The advice that the programmer gives to CLIM on how to assemble the table consists of the following:

- How to place the contents of the cell within the cell (such as centered vertically, flush-left, and so forth) The possibilities for this advice are described below.
- Optionally, how much additional space to insert between columns and between rows of the table.
- Optionally, whether to make all columns the same size.

The advice describing how to place the contents of the cell within the cell consists of two pieces—how to constrain the cell contents in the horizontal direction, and how to constrain them in the vertical direction.

17.2 Table Formatting Functions

⇒ `formatting-table` *(*`&optional` *stream* `&key` *x-spacing y-spacing multiple-columns multiple-columns-x-spacing equalize-column-widths (move-cursor* `t`*) record-type)* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be done in a tabular format. This must be used in conjunction with `formatting-row` or `formatting-column`, and `formatting-cell`. The table is placed so that its upper left corner is at the current text cursor position of *stream*. If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the last cell of the table.

The returned value is the output record corresponding to the table.

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

x-spacing specifies the number of units of spacing to be inserted between columns of the table; the default is the width of a space character in the current text style. *y-spacing* specifies the number of units of spacing to be inserted between rows in the table; the default is the default vertical spacing of the stream. Possible values for these two options option are:

- An integer—a size in the current units to be used for spacing.
- A string or character—the spacing is the width or height of the string or character in the current text style.
- A function—the spacing is the amount of horizontal or vertical space the function would consume when called on the stream.
- A list—the list is of the form `(`*number unit*`)`, where *unit* is one of `:point`, `:pixel`, `:mm`, `:character`, or `:line`. When *unit* is `:character`, the width of an "M" in the current text style is used as the width of one character.

multiple-columns is either `nil`, `t`, or an integer. If it is `t` or an integer, the table rows will be broken up into a multiple columns. If it is `t`, CLIM will determine the optimal number of columns. If it is an integer, it will be interpreted as the desired number of columns. *multiple-columns-x-spacing* has the same format as *x-spacing*. It controls the spacing between the multiple columns. It defaults to the value of the *x-spacing* option.

When the boolean *equalize-column-widths* is true, CLIM will make all of the columns have the same width (the width of the widest cell in any column in the entire table).

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

⇒ `formatting-row` *(*`&optional` *stream* `&key` *record-type)* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be grouped into a table row. All of the output performed by *body* becomes the contents of one row. This must be used inside of `formatting-table`, and in conjunction with `formatting-cell`.

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

Once a table has had a row added to it via `formatting-row`, no columns may be added to it.

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

⇒ `formatting-column` *(*`&optional` *stream* `&key` *record-type)* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be grouped into a table column. All of the output performed by *body* becomes the contents of one column. This must be used inside of `formatting-table`, and in conjunction with `formatting-cell`.

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

Once a table has had a column added to it via `formatting-column`, no rows may be added to it.

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

⇒ `formatting-cell` *(*`&optional` *stream* `&key` *(align-x* `':left`*) (align-y* `':baseline`*) min-width min-height record-type)* `&body` *body* [*Macro*]

Controls the output of a single cell inside a table row or column, or of a single item inside `formatting-`

`item-list`. All of the output performed by *body* becomes the contents of the cell.

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

align-x specifies how the output in a cell will be aligned relative to other cells in the same table column. The default, `:left`, causes the cells to be flush-left in the column. The other possible values are `:right` (meaning flush-right in the column) and `:center` (meaning centered in the column). Each cell within a column may have a different alignment; thus it is possible, for example, to have centered legends over flush-right numeric data.

align-y specifies how the output in a cell will be aligned vertically. The default, `:baseline`, causes textual cells to be aligned along their baselines and graphical cells to be aligned at the bottom. The other possible values are `:bottom` (align at the bottom of the output), `:top` (align at the top of the output), and `:center` (center the output in the cell).

min-width and *min-height* are used to specify minimum width or height of the cell. The default, `nil`, causes the cell to be only as wide or high as is necessary to contain the cell's contents. Otherwise, *min-width* and *min-height* are specified in the same way as the `:x-spacing` and `:y-spacing` arguments to `formatting-table`.

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

⇒ `formatting-item-list` *(*`&optional` *stream* `&key` *x-spacing y-spacing n-columns n-rows stream-width stream-height max-width max-height initial-spacing (move-cursor* `t`*) record-type)* `&body` *body* [*Macro*]

Binds the local environment in such a way that the output of *body* will be done in an item list (that is, menu) format. This must be used in conjunction with `formatting-cell`, which delimits each item. The item list is placed so that its upper left corner is at the current text cursor position of *stream*. If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the last cell of the item list.

"Item list output" is more strictly defined as: each row of the item list consists of a single cell. Rows are placed with the first row on top, and each succeeding row has its top aligned with the bottom of the previous row (plus the specified *y-spacing*). Multiple rows and columns are constructed after laying the item list out in a single column. Item list output takes place in a normalized ***+y***-downward coordinate system.

The returned value is the output record corresponding to the table.

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

x-spacing specifies the number of units of spacing to be inserted between columns of the item list; the default is the width of a `#\Space` character in the current text style. *y-spacing* specifies the number of units of spacing to be inserted between rows in the item list; the default is default vertical spacing of the stream. The format of these arguments is as for `formatting-table`.

When the boolean *equalize-column-widths* is *true*, CLIM will make all of the columns have the same width (the width of the widest cell in any column in the entire item list).

n-columns and *n-rows* specify the number of columns or rows in the item list. The default for both is `nil`, which causes CLIM to pick an aesthetically pleasing layout, possibly constrained by the other options. If both *n-columns* and *n-rows* are supplied and the item list contains more elements than will fit according to the specification, CLIM will format the item list as if *n-rows* were supplied as `nil`.

max-width and *max-height* constrain the layout of the item list. CLIM will not make the item list any wider than *max-width*, unless it is overridden by *n-rows*. It will not make the item list any taller than *max-height*, unless it is overridden by *n-columns*.

`formatting-item-list` normally spaces items across the entire width of the stream. When *initial-spacing* is *true*, it inserts some whitespace (about half as much space as is between each item) before the first item on each line. When it is *false* (the default), the initial whitespace is not inserted.

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

⇒ `format-items` *items* `&key` *stream printer presentation-type x-spacing y-spacing n-columns n-rows max-width max-height cell-align-x cell-align-y initial-spacing (move-cursor* `t`*) record-type* [*Function*]

This is a function interface to the item list formatter. The elements of the sequence *items* are formatted as separate cells within the item list.

stream is an output recording stream to which output will be done. It defaults to `*standard-output*`.

printer must be a function that takes two arguments, an item and a stream, and outputs the item on the stream. *printer* has dynamic extent. The default for *printer* is `prin1`.

presentation-type is a presentation-type. When *printer* is not supplied, the items will be printed as if *printer* were

```
#'(lambda (item stream)
    (present item presentation-type :stream stream))
```

When *printer* is supplied, each item will be enclosed in a presentation whose type is *presentation-type*.

x-spacing, *y-spacing*, *n-columns*, *n-rows*, *max-width*, *max-height*, *initial-spacing*, and *move-cursor* are as for `formatting-item-list`.

cell-align-x and *cell-align-y* are used to supply `:align-x` and `:align-y` to an implicitly used `formatting-cell`.

record-type is as for `formatting-item-list`.

17.3 The Table and Item List Formatting Protocols

Both table and item list formatting is implemented on top of the basic output recording protocol, using `with-new-output-record` to specify the appropriate type of output record. For example, `formatting-table` first collects all the output that belongs in the table into a collection of row, column, and cell output records, all of which are children of a single table output record. During this phase, `stream-drawing-p` is bound to `nil` and `stream-recording-p` is bound to `t`. When all the output has been generated, the table layout constraint solver (`adjust-table-cells` or `adjust-item-list-cells`) is called to compute the table layout, taking into account such factors as the widest cell in a given column. If the table is to be split into multiple columns, `adjust-multiple-columns` is now called. Finally, the table output record is positioned on the stream at the current text cursor position and then displayed by calling `replay` on the table (or item list) output record.

17.3.1 Table Formatting Protocol

Any output record class that implements the following generic functions is said to support the table formatting protocol.

In the following subsections, the term "non-table output records" will be used to mean any output record that is not a table, row, column, cell, or item list output record. When CLIM "skips over intervening non-table output records", this means that it will bypass all the output records between two such table output records (such as a table and a row, or a row and a cell) that are not records of those classes (most notably, presentation output records). CLIM implementations are encouraged to detect invalid nesting of table output records, such as a row within a row, a cell within a cell, or a row within a cell. Note that this does not prohibit the nesting of calls to `formatting-table`, it simply requires that programmers include the inner table within one of the cells of the outer table.

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

The protocol class that represents tabular output records; a subclass of `output-record`. If you want to create a new class that behaves like a table output record, it should be a subclass of `table-output-record`. Subclasses of `table-output-record` must obey the table output record protocol.

⇒ `table-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *table output record*, otherwise returns *false*.

⇒ `:x-spacing` [*Init Arg*]

⇒ `:y-spacing` [*Init Arg*]

⇒ `:multiple-columns-x-spacing` [*Init Arg*]

⇒ `:equalize-column-widths` [*Init Arg*]

All subclasses of `table-output-record` must handle these initargs, which are used to specify, respectively, the x and y spacing, the multiple column x spacing, and equal-width columns attributes of the table.

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

The instantiable class of output record that represents tabular output. Its children will be a sequence of either rows or columns, with presentation output records possibly intervening. This is a subclass of `table-output-record`.

⇒ `map-over-table-elements` *function table-record type* [*Generic Function*]

Applies *function* to all the rows or columns of *table-record* that are of type *type*. *type* is either `:row`, `:column`, or `:row-or-column`. *function* is a function of one argument, an output record; it has dynamic extent. `map-over-table-elements` is responsible for ensuring that rows, columns, and cells are properly nested. It must skip over intervening non-table output record structure, such as presentations.

⇒ `adjust-table-cells` *table-record* [*Generic Function*]

This function is called after the tabular output has been collected, but before it has been replayed. The method on `standard-table-output-record` implements the usual table layout constraint solver (described above) by moving the rows or columns of the tables and the cells within the rows or columns.

⇒ `adjust-multiple-columns` *table-record* [*Generic Function*]

This is called after `adjust-table-cells` to account for the case where the programmer wants the table to have multiple columns.

17.3.2 Row and Column Formatting Protocol

Any output record class that implements the following generic functions is said to support the row (or column) formatting protocol.

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

The protocol class that represents one row in a table; a subclass of `output-record`. If you want to create a new class that behaves like a row output record, it should be a subclass of `row-output-record`. Subclasses of `row-output-record` must obey the row output record protocol.

⇒ `row-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *row output record*, otherwise returns *false*.

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

The instantiable class of output record that represents a row of output within a table. Its children will be a sequence of cells, and its parent (skipping intervening non-tabular records such as presentations) will be a table output record. This is a subclass of `row-output-record`.

⇒ `map-over-row-cells` *function row-record* [*Generic Function*]

Applies *function* to all the cells in the row *row-record*, skipping intervening non-table output record structure. *function* is a function of one argument, an output record corresponding to a table cell within the row; it has dynamic extent.

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

The protocol class that represents one column in a table; a subclass of `output-record`. If you want to create a new class that behaves like a column output record, it should be a subclass of `column-output-record`. Subclasses of `column-output-record` must obey the column output record protocol.

⇒ `column-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *column output record*, otherwise returns *false*.

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

The instantiable class of output record that represent a column of output within a table. Its children will be a sequence of cells, and its parent (skipping intervening non-tabular records such as presentations) will be a table output record; presentation output records may intervene. This is a subclass of `column-output-record`.

⇒ `map-over-row-cells` *function column-record* [*Generic Function*]

Applies *function* to all the cells in the column *column-record*, skipping intervening non-table output record structure. *function* is a function of one argument, an output record corresponding to a table cell within the column; it has dynamic extent.

17.3.3 Cell Formatting Protocol

Any output record class that implements the following generic functions is said to support the cell formatting protocol.

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

The protocol class that represents one cell in a table or an item list; a subclass of `output-record`. If you want to create a new class that behaves like a cell output record, it should be a subclass of `cell-output-record`. Subclasses of `cell-output-record` must obey the cell output record protocol.

⇒ `cell-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *cell output record*, otherwise returns *false*.

⇒ `:align-x` [*Init Arg*]

⇒ `:align-y` [*Init Arg*]

⇒ `:min-width` [*Init Arg*]

⇒ `:min-height` [*Init Arg*]

All subclasses of `cell-output-record` must handle these initargs, which are used to specify, respectively, the x and y alignment, and the minimum width and height attributes of the cell.

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

The instantiable class of output record that represent a single piece of output within a table row or column, or an item list. Its children will either be presentations, or output records that represent displayed output. This is a subclass of `cell-output-record`.

⇒ `cell-align-x` *cell* [*Generic Function*]
⇒ `cell-align-y` *cell* [*Generic Function*]
⇒ `cell-min-width` *cell* [*Generic Function*]
⇒ `cell-min-height` *cell* [*Generic Function*]

These functions return, respectively, the x and y alignment and minimum width and height of the *cell output record cell*.

17.3.4 Item List Formatting Protocol

⇒ `item-list-output-record` [*Protocol Class*]

The protocol class that represents an item list; a subclass of `output-record`. If you want to create a new class that behaves like an item list output record, it should be a subclass of `item-list-output-record`. Subclasses of `item-list-output-record` must obey the item list output record protocol.

⇒ `item-list-output-record-p` *object* [*Function*]

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

⇒ `:x-spacing` [*Init Arg*]

⇒ `:y-spacing` [*Init Arg*]

⇒ `:initial-spacing` [*Init Arg*]

⇒ `:n-rows` [*Init Arg*]

⇒ `:n-columns` [*Init Arg*]

⇒ `:max-width` [*Init Arg*]

⇒ `:max-height` [*Init Arg*]

All subclasses of `item-list-output-record` must handle these initargs, which are used to specify, respectively, the x and y spacing, the initial spacing, the desired number of rows and columns, and maximum width and height attributes of the item list.

⇒ `standard-item-list-output-record` [*Class*]

The output record that represents item list output. Its children will be a sequence of cells, with presentations possibly intervening. This is a subclass of `item-list-output-record`.

⇒ `map-over-item-list-cells` *function item-list-record* [*Generic Function*]

Applies *function* to all of the cells in *item-list-record*. `map-over-item-list-cells` must skip over intervening non-table output record structure, such as presentations. *function* is a function of one argument, an output record corresponding to a cell in the item list; it has dynamic extent.

⇒ `adjust-item-list-cells` *item-list-record* [*Generic Function*]

This function is called after the item list output has been collected, but before the record has been replayed. The method on `standard-item-list-output-record` implements the usual item list layout constraint solver.

Chapter 18

Graph Formatting

CLIM provides a mechanism for arranging arbitrary output in a graph. The following code produces the graph shown in Figure 18.1.

```
(defun graph-test (stream &optional (orientation :horizontal))
  (fresh-line stream)
  (macrolet ((make-node (&key name children)
               `(list* ,name ,children)))
    (flet ((node-name (node)
             (car node))
           (node-children (node)
             (cdr node)))
      (let* ((2a (make-node :name "2A"))
             (2b (make-node :name "2B"))
             (2c (make-node :name "2C"))
             (1a (make-node :name "1A" :children (list 2a 2b)))
             (1b (make-node :name "1B" :children (list 2b 2c)))
             (root (make-node :name "0" :children (list 1a 1b))))
        (format-graph-from-roots
          (list root)
          #'(lambda (node s)
              (write-string (node-name node) s))
          #'node-children
          :orientation orientation
          :stream stream)))))
```

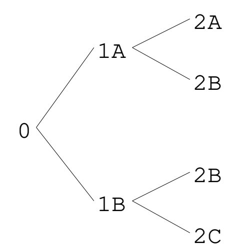


Figure 18.1: Example of graph formatting.

18.1 Graph Formatting Functions

⇒ `format-graph-from-roots` *root-objects object-printer inferior-producer* `&key` *stream orientation cutoff-depth merge-duplicates duplicate-key duplicate-test generation-separation within-generation-separation arc-drawer arc-drawing-options graph-type (move-cursor* `t`*)* [*Function*]

Draws a graph whose roots are specified by the sequence *root-objects*. The nodes of the graph are displayed by calling the function *object-printer*, which takes two arguments, the node to display and a stream. *inferior-producer* is a function of one argument that is called on each node to produce a sequence of inferiors (or `nil` if there are none). Both *object-printer* and *inferior-producer* have dynamic extent.

The output from graph formatting takes place in a normalized $+y$-downward coordinate system. The graph is placed so that the upper left corner of its bounding rectangle is at the current text cursor position of *stream*. If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the graph.

The returned value is the output record corresponding to the graph.

stream is an output recording stream to which output will be done. It defaults to `*standard-output*`.

orientation may be either `:horizontal` (the default) or `:vertical`. It specifies which way the graph is oriented. CLIM implementations are permitted to extend the values of *orientation*, for example, adding `:right` or `:left` to distinguish between left-to-right or right-to-left layouts.

cutoff-depth specifies the maximum depth of the graph. It defaults to `nil`, meaning that there is no cutoff depth. Otherwise it must be an integer, meaning that no nodes deeper than *cutoff-depth* will be formatted or displayed.

If the boolean *merge-duplicates* is *true*, then duplicate objects in the graph will share the same node in the display of the graph. That is, when *merge-duplicates* is *true*, the resulting graph will be a tree.

If *merge-duplicates* is *false* (the default), then duplicate objects will be displayed in separate nodes. *duplicate-key* is a function of one argument that is used to extract the node object component used for duplicate comparison; the default is `identity`. *duplicate-test* is a function of two arguments that is used to compare two objects to see if they are duplicates; the default is `eql`. *duplicate-key* and *duplicate-test* have dynamic extent.

generation-separation is the amount of space to leave between successive generations of the graph; the default is 20. *within-generation-separation* is the amount of space to leave between nodes in the same generation of the graph; the default is 10. *generation-separation* and *within-generation-separation* are specified in the same way as the *inter-row-spacing* argument to `formatting-table`.

arc-drawer is a function of seven positional and some unspecified keyword arguments that is responsible for drawing the arcs from one node to another; it has dynamic extent. The positional arguments are the stream, the "from" node, the "to" node, the "from" x and y position, and the "to" x and y position. The keyword arguments are typically line drawing options, such as for `draw-line*`. If *arc-drawer* is unsupplied, the default behavior is to draw a thin line from the "from" node to the "to" node using `draw-line*`.

graph-type is a keyword that specifies the type of graph to draw. CLIM implementations must support `:tree`, `:directed-graph` (and its synonym `:digraph`), and `:directed-acyclic-graph` (and its synonym `:dag`). *graph-type* defaults to `:tree` when *merge-duplicates* is *true*, otherwise it defaults to `:digraph`. Typically, different graph types will use different output record classes and layout engines to lay out the graph. However, it is permissible for all of the required graph types to use exactly the same graph layout engine.

18.2 The Graph Formatting Protocols

Graph formatting is implemented on top of the basic output recording protocol, using `with-new-output-record` to specify the appropriate type of output record. For example, `format-graph-from-roots` first collects all the output that belongs in the graph into a collection of graph node output records by calling `generate-graph-nodes`. All of the graph node output records are descendents of a single graph output record. During this phase, `stream-drawing-p` is bound to `nil` and `stream-recording-p` is bound to `t`. When all the output has been generated, the graph layout code (`layout-graph-nodes` and `layout-graph-edges`) is called to compute the graph layout. Finally, the graph output record is positioned on the stream at the current text cursor position and then displayed by calling `replay` on the graph output record.

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

The protocol class that represents a graph; a subclass of `output-record`. If you want to create a new class that behaves like a graph output record, it should be a subclass of `graph-output-record`. Subclasses of `graph-output-record` must obey the graph output record protocol.

⇒ `graph-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *graph output record*, otherwise returns *false*.

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

The instantiable class of output record that represents a graph. Its children will be a sequence graph nodes. This is a subclass of `graph-output-record`.

⇒ `:orientation` [*Init Arg*]

⇒ `:center-nodes` [*Init Arg*]

⇒ `:cutoff-depth` [*Init Arg*]

⇒ `:merge-duplicates` [*Init Arg*]

⇒ `:generation-separation` [*Init Arg*]

⇒ `:within-generation-separation` [*Init Arg*]

⇒ `:hash-table` [*Init Arg*]

All the graph output record must handle these seven initargs, which are used to specify, respectively, the orientation, node centering, cutoff depth, merge duplicates, generation and within-generation spacing, and the node hash table of a graph output record.

⇒ `*graph-type-record-type-alist*` [*Variable*]

An alist that map from a graph type to the name of the graph output record class that implements the graph type.

⇒ `graph-root-nodes` *graph-record* [*Generic Function*]

Returns a sequence of the graph node output records corresponding to the root objects for the graph output record *graph-record*.

⇒ `(setf graph-root-nodes)` *roots graph-record* [*Generic Function*]

Sets the root nodes of *graph-record* to *roots*.

⇒ `generate-graph-nodes` *graph-record stream root-objects object-printer inferior-producer* `&key` *duplicate-key duplicate-test* [*Generic Function*]

This function is responsible for generating all of the graph node output records of the graph. *graph-record* is the graph output record, and *stream* is the output stream. The graph node output records are generating by calling the object printer on the root objects, then recursively calling the inferior producer on the root objects and calling the object printer on all inferiors. After all of the graph node output records have been generated, the value of `graph-root-nodes` of *graph-record* must be set to be a sequence of the those graph node output records that correspond to the root objects.

root-objects, *object-printer*, *inferior-producer*, *duplicate-key*, and *duplicate-test* are as for `format-graph-from-roots`.

⇒ `layout-graph-nodes` *graph-record stream* [*Generic Function*]

This function is responsible for laying out the nodes in the graph. It is called after the graph output has been collected, but before the graph record has been displayed. The method on `standard-graph-output-record` implements the usual graph layout constraint solver.

⇒ `layout-graph-edges` *graph-record stream arc-drawer arc-drawing-options* [*Generic Function*]

This function is responsible for laying out the edges in the graph. It is called after the graph nodes have been layed out, but before the graph record has been displayed. The method on `standard-graph-output-record` simply causes thin lines to be drawn from each node to all of its children.

⇒ `graph-node-output-record` [*Protocol Class*]

The protocol class that represents a node in graph; a subclass of `output-record`. If you want to create a new class that behaves like a graph node output record, it should be a subclass of `graph-node-output-record`. Subclasses of `graph-node-output-record` must obey the graph node output record protocol.

⇒ `graph-node-output-record-p` *object* [*Function*]

Returns *true* if *object* is a *graph node output record*, otherwise returns *false*.

⇒ `standard-graph-node-output-record` [*Class*]

The instantiable class of output record that represents a graph node. Its parent will be a graph output record. This is a subclass of `graph-node-output-record`.

⇒ `graph-node-parents` *graph-node-record* [*Generic Function*]

Returns a sequence of the graph node output records whose objects are "parents" of the object corresponding to the graph node output record *graph-node-record*. Note that this is not the same as `output-record-parent`, since `graph-node-parents` can return output records that are not the parent records of *graph-node-record*.

⇒ `(setf graph-node-parents)` *parents graph-node-record* [*Generic Function*]

Sets the parents of *graph-node-record* to be *parents*. *parents* must be a list of graph node records.

⇒ `graph-node-children` *graph-node-record* [*Generic Function*]

Returns a sequence of the graph node output records whose objects are "children" of the object corresponding to the graph node output record *graph-node-record*. Note that this is not the same as `output-record-children`, since `graph-node-children` can return output records that are not child records of *graph-node-record*.

⇒ `(setf graph-node-children)` *children graph-node-record* [*Generic Function*]

Sets the children of *graph-node-record* to be *children*. *children* must be a list of graph node records.

⇒ `graph-node-object` *graph-node-record* [*Generic Function*]

Returns the object that corresponds to the graph node output record *graph-node-record*. It is permissible for this function to work correctly only while inside of the call to `format-graph-from-roots`. It is unspecified what result will be returned outside of `format-graph-from-roots`. This restriction is permitted so that CLIM is not required to capture application objects that might have dynamic extent.

Chapter 19

Bordered Output

CLIM provides a mechanism for surrounding arbitrary output with some kind of a border. The programmer annotates some output-generating code with an advisory macro that describes the type of border to be drawn. The following code produces the output shown in Figure 19.1.

For example, the following produces a piece of output surrounded by a rectangle.

```
(defun border-test (stream)
  (fresh-line stream)
  (surrounding-output-with-border (stream :shape :rectangle)
    (format stream "This is some output with a rectangular border"))
  (terpri stream) (terpri stream)
  (surrounding-output-with-border (stream :shape :drop-shadow)
    (format stream "This has a drop-shadow under it"))
  (terpri stream) (terpri stream)
  (surrounding-output-with-border (stream :shape :underline)
    (format stream "And this output is underlined")))
```

⇒ `surrounding-output-with-border` *(*`&optional` *stream* `&key` *shape (move-cursor* `t`*))* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be surrounded by a border of the specified shape. Every implementation must support the shapes `:rectangle` (the default), `:drop-shadow`, and `:underline`. `:rectangle` draws a rectangle around the bounding rectangle of the output. `:drop-shadow` draws a "drop shadow" around the lower right edge of the bounding rectangle of the output. `:underline` draws a thin line along the baseline of all of the text in the output, but does not draw anything underneath non-textual output.

If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the bordered output.

stream is an output recording stream to which output will be done. The *stream* argument is not evaluated,

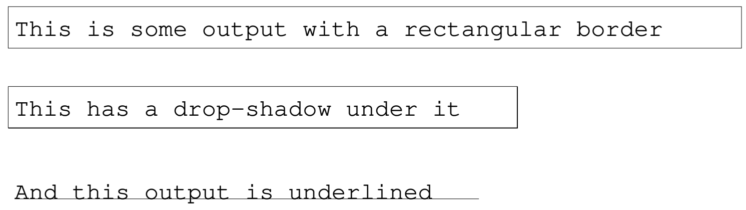


Figure 19.1: Examples of bordered output.

and must be a symbol that is bound to a stream. If *stream* is `t` (the default), `*standard-output*` is used. *body* may have zero or more declarations as its first forms.

There are several strategies for implementing borders. One strategy is to create a "border output record" that contains the output records produced by the output of *body*, plus one or more output records that represent the border. Another strategy might be to arrange to call the border drawer at the approriate times without explicitly recording it.

⇒ `define-border-type` *shape arglist* `&body` *body* [*Macro*]

Defines a new kind of border named *shape*. *arglist* must be a subset of the "canonical" arglist below (using `string-equal` to do the comparison):
(`&key` *stream record left top right bottom)*

body is the code that actually draws the border. It has lexical access to `stream`, `record`, `left`, `top`, `right`, and `bottom`, which are respectively, the stream being drawn on, the output record being surrounded, and the coordinates of the left, top, right, and bottom edges of the bounding rectangle of the record. *body* may have zero or more declarations as its first forms.

Chapter 20

Text Formatting

20.1 Textual List Formatting

⇒ `format-textual-list` *sequence printer* `&key` *stream separator conjunction* [*Function*]

Outputs the sequence of items in *sequence* as a "textual list". For example, the list `(1 2 3 4)` might be printed as

```
1, 2, 3, and 4
```

printer is a function of two arguments: an element of the sequence and a stream; it has dynamic extent. It is called to output each element of the sequence.

stream specifies the output stream. The default is `*standard-output*`.

The *separator* and *conjunction* arguments provide control over the appearance of each element of the sequence and over the separators used between each pair of elements. *separator* is a string that is output after every element but the last one; the default for *separator* is `", "` (that is, a command followed by a space). *conjunction* is a string that is output before the last element. The default is `nil`, meaning that there is no conjunction. Typical values for *conjunction* are the strings `"and"` and `"or"`.

20.2 Indented Output

⇒ `indenting-output` *(stream indentation* `&key` *(move-cursor* `t`*))* `&body` *body* [*Macro*]

Binds *stream* to a stream that inserts whitespace at the beginning of each line of output produced by *body*, and then writes the indented output to the stream that is the original value of *stream*.

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.

indentation specifies how much whitespace should be inserted at the beginning of each line. It is specified in the same way as the `:x-spacing` option to `formatting-table`.

If the boolean *move-cursor* is *true* (the default), CLIM moves the cursor to the end of the table.

Programmers using `indenting-output` should begin the body with a call to *fresh-line* (or some equivalent) to position the stream to the indentation initially. There is a restriction on interactions between `indenting-output` and `filling-output` such that a call to `indenting-output` should appear outside of a call to `filling-output`. Implementations are permitted to relax this restriction, but uses of `indenting-output` inside of `filling-output` may not be portable.

20.3 Filled Output

⇒ `filling-output` *(stream* `&key` *fill-width break-characters after-line-break after-line-break-initially)* `&body` *body* [*Macro*]

Binds *stream* to a stream that inserts line breaks into the textual output written to it (by such functions as `write-char` and `write-string`) so that the output is usually no wider then *fill-width*. The filled output is then written on the original stream.

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

fill-width specifies the width of filled lines, and defaults to 80 characters. It is specified the same way as the `:x-spacing` option for `formatting-table`.

"Words" are separated by the characters specified in the list *break-characters*. When a line is broken to prevent wrapping past the end of a line, the line break is made at one of these separators. That is, `filling-output` does not split "words" across lines, so it might produce output wider than *fill-width*.

after-line-break specifies a string to be sent to *stream* after line breaks; the string appears at the beginning of each new line. The string must not be wider than *fill-width*.

If the boolean *after-line-break-initially* is *true*, then the *after-line-break* text is to be written to *stream* before executing *body*, that is, at the beginning of the first line. The default is *false*.

Chapter 21

Incremental Redisplay

Incremental Redisplay is a facility to allow you to change the output in an output history (and hence, on the screen or other output device). It allows you to redisplay pieces of the existing output differently, under your control.

It is "incremental" in the sense that the system will try to minimize the changes to the output device in order to convert the output to follow your specification.

There are two different ways to do incremental redisplay.

Issue: This whole chapter still needs serious work. — SWM

Issue: Prefer using parent/children terminology to superior/inferiors. — SWM

The first is to call `redisplay` on an output record. This essentially tells the system to start that output record over from scratch. It compares the results with the existing output and tries to do minimal redisplay. The `updating-output` form allows you to assist the system by informing it that entire branches of the output history are known not to have changed. `updating-output` also allows you to communicate the fact that a piece of the output record hierarchy has moved.

Issue: What does "moved" mean? Does it mean "changed its sibling ordering" or can it also mean "changed its ancestry"? The former is a lot easier (and perhaps cleaner and more intuitive) than the latter. The whole "move" concept also points out that there are really three IDs (unique, cache and position) but only two get documented and the third is not only behind the scenes but very magical. — DCPL

Issue: I think that the "position id" stuff should be flushed from the spec, but implementations should not be forbidden from implementing it. It is very difficult to specify closely, and is altogether too magic, even though it is useful. This would also kill the last remaining strong reason for storing transformations explicitly in output records. — SWM

The second way to do incremental redisplay is to manually do the updates to the output history, and then call `inferior-output-record-changed` on an output record. This allows the system to propagate the changes up the output record history hierarchy and allow parent output records to readjust themselves to account for your changes.

Each is appropriate under different circumstances. `redisplay` is often easier to code, and more useful in cases where there might be large changes between two passes, or where you have little idea as to what the changes might be. `inferior-output-record-changed` can be more efficient for small changes at the bottom of the output record hierarchy, or in cases where you are well informed as to the specific changes necessary and can help the system out.

The behavior of incremental redisplay can be modified or customized by replacing the generic functions that make up the Incremental Redisplay protocol.

21.1 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.

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

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
```

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

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

for the form after the call to `sleep`), Incremental Redisplay 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 the discussion of Application Building Tools elsewhere in this document for examples of how to use Incremental Redisplay automatically within your application.

21.2 Standard Programmer Interface

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

Introduces a caching point for incremental redisplay. 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.

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.

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 *copy-cache-value* is *true*, then the supplied cache value will be copied using `copy-seq` before it is stored in the output record. The default for *copy-cache-value* is *false*.

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. The default for *all-new* is *false*.

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.

⇒ `redisplay` *record stream* [*Function*]

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

⇒ `redisplay-output-record` *record stream* `&optional` *x y parent-x parent-y* [*Generic Function*]

Issue: The coordinate system stuff affected by the x/y and parent-x/y arguments is entirely bogus. My 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.

The optional arguments can be used to specify where on the *stream* the output record should be redisplayed. *x* and *y* represent where the cursor should be, relative to (`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

⇒ `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`. Subclasses of `updating-output-record` must obey the updating output record protocol.

⇒ `updating-output-record-p` *object* [*Function*]

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

⇒ `:unique-id` [*Init Arg*]

⇒ `:id-test` [*Init Arg*]

⇒ `:cache-value` [*Init Arg*]

⇒ `:cache-test` [*Init Arg*]

⇒ `:fixed-position` [*Init Arg*]

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.

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

`inferior-output-record-changed` assumes that the output history already reflects the changes made, but that no output (screen, or hardcopy, or ...) has been done yet. (Some) parent will (eventually) do the appropriate updates. You must pass the "differences list" (output in the style of `compute-differences`) to `inferior-output-record-changed`. If you don't know the optimized form of the differences, then you must call `compute-differences` before you update the output record history, and pass the results to `inferior-output-record-changed`.

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

old-position and *old-extent* are where *child* was before it was moved.

This is the protocol that must be supported by any output records that wish to be compatible with incremental redisplay.

Matching and automatic adjustment:

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

Returns *true* if record matches the supplied init-args, otherwise returns *false*.

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

Finds a child of *record* matching the *record-type* and *init-args* supplied. *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.

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

To be supplied.

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

To be supplied.

⇒ `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.

The following functions are called by `updating-output`.

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

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

⇒ `find-cached-output-record` *record use-old-elements record-type* `&rest` *init-args* `&key` *unique-*

id unique-id-test `&allow-other-keys` [*Generic Function*]

Finds a previously cached child matching *record-type*, *init-args*, *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.

The following functions are called by `redisplay-output-record`.

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

The contract of `new-output-records` is to modify an output history to what the record should look like now. It must record all of the old positions in old-extent, old-start-position, and old-elements.

It recurses down the descendents doing `new-output-records` on each child.

`new-output-records` of an output record of type `updating-output-record` runs the 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-differences` *record* `&optional` *check-overlapping (offset-x* `0`*) (offset-y* `0`*) (old-offset-x* `0`*) (old-offset-y* `0`*)* [*Generic Function*]

This operation compares the old and new versions of *record* and returns the differences as 5 multiple values. The values returned are *erases*, *draws*, *moves*, *erase-overlapping*, and *move-overlapping*. Each is a list of lists of the form:

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

The following functions are called by `inferior-output-record-changed`.

⇒ `augment-draw-set` *record* [*Generic Function*]

To be supplied.

⇒ `propagate-inferior-output-record-changes-p` *record child mode old-position old-extent* [*Generic Function*]

`propagate-inferior-output-record-changes-p` is a predicate which 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-child-position* `(output-record-old-start-position child)`*) (old-child-extent* `(output-record-old-extent child)`*) erases moves draws erase-overlapping move-overlapping check-overlapping* [*Generic Function*]

To be supplied.

21.4 Incremental Redisplay Stream Protocol

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

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

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

erases, *moves*, *draws*, *erase-overlapping*, and *move-overlapping* are differences lists as returned by `compute-differences`.

`incremental-redisplay` can be called on any stream that can set the cursor position and replace existing output.

21.5 Random Comments

Since we allow incremental redisplay to work on any output record, `updating-output` is just a hint. It can tell us about two different kinds of things: (1) whether things have changed at all and (2) what this new output record corresponds to in the old hierarchy.

There are four ways to determine the output record in the current history that *body* corresponds to. The first is if the programmer passes in `:output-record`. The rest all rely on `:unique-id` and `:id-test`. The tree of `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 P with `:parent-cache`, then we try to find a corresponding output record with the matching unique-id in the cache belonging to P. If neither `:output-record` nor `:parent-cache` is provided, then we look for the unique-id in the immediately dynamically enclosing `updating-output`. Finally, if all else fails, we use the unique-id to find the output record that is an child of `(output-recording-stream-output-record` *stream*`)`.

Alternatively, you can specify `:all-new t`, which guarantees that nothing in this output record (or in any of its descendants) will match anything on the screen, so don't even bother trying.

Once we have a correspondence, we can use the cache-value and cache-test to tell us whether this output record has changed. If it hasn't, and it's on the screen, we can just move the bits on the screen around.

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 C), 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-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-read-char-no-hang` *stream* [*Generic Function*]

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

⇒ `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`. Subclasses of `extended-input-stream` must obey the extended input stream protocol.

⇒ `extended-input-stream-p` *object* [*Function*]

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

⇒ `:input-buffer` [*Init Arg*]

⇒ `:pointer` [*Init Arg*]

⇒ `:text-cursor` [*Init Arg*]

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 characters and event objects).

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

Returns the list of pointer objects corresponding to the pointing devices of the port associated with *stream*. This function returns objects that reveal CLIM's internal state; do not modify those objects.

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

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

Returns (or sets) the pointer object corresponding to the primary pointing device of the console.

Note: CLIM currently supports only a single pointer for any port. Therefore, the length of the list returned by `stream-pointers` will always be one, and `stream-primary-pointer` will always return an object that is the only element of that list.

⇒ `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 the `stream-primary-pointer` of the stream.

⇒ `(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*`.

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

Sets the "input focus" to the *extended input stream stream* and returns the old input focus as its value.

⇒ `stream-restore-input-focus` *stream old-focus* [*Generic Function*]

Restores the "input focus" of the *extended input stream stream* to *old-focus*.

⇒ `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.

⇒ `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.

`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 invoke a gesture preprocessing mechanism on every gesture. For a discussion of this facility, see the Section 22.6.

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`.

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).

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.

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`.

22.3 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 comment 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* [*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 `nil`.

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

⇒ `add-gesture-name` *name type gesture-spec* [*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*.

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 include `on-genera:help` and `on-genera: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 :keystroke (#\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*.

Issue: The problem with all of the stuff above is that gesture names are port-specific, but there is no port argument anywhere. — SWM

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` 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`. Subclasses of `pointer` must obey the pointer protocol. Members of this class are mutable.

⇒ `pointerp` *object* [*Function*]

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

⇒ `:port` [*Init Arg*]

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-port` *pointer* [*Generic Function*]

Returns the port with which the *pointer pointer* is associated.

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

Returns the state of the buttons of the *pointer pointer*. This is represented in the same as `pointer-event-button`.

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

Returns the x and y position of the *pointer pointer* as two values.

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

Sets the x and y position of the *pointer pointer* to the specified 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*.

22.5 Pointer Tracking

⇒ `tracking-pointer` *(sheet* `&key` *pointer multiple-windows transformp context-type)* `&body` *body* [*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 of a pointer button on 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, `(stream-primary-pointer` *sheet*`)`.

When the boolean *multiple-window* 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. This is only meaningful when *sheet* is an output recording stream. *context-type* defaults to `t`, meaning that all presentations are visible.

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:

- `: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.)
- `: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.)

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

- `: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*.)

  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 if applicable, otherwise the `:pointer-button-press` clause will run.

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

- `: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.

⇒ `drag-output-record` *stream output-record* `&key` *repaint erase feedback finish-on-release* [*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.

⇒ `dragging-output` *(*`&optional` *stream* `&key` *repaint finish-on-release)* `&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 and *finish-on-release* are as for `drag-output-record`.

22.6 Gesture Preprocessing

Every gesture that goes through `stream-read-gesture` gets processed before it is returned. This allows applications to intercept gestures and alter the flow of control. Gesture processing is done via a gesture table that associates each gesture with a function of two arguments, *stream* and *gesture*, which is applied when the gesture is read.

Issue: So what is "gesture preprocessing" anyway? It should surely specialize on the stream, or the application frame, or both. — SWM

⇒ `gesture-processing-handler` *gesture* [*Function*]
⇒ `(setf gesture-processing-handler)` *gesture* [*Function*]

Manipulate the gesture processing handler for a particular gesture. All pointer button presses are considered equivalent at this layer; to modify the handler for pointer button presses, specify `:pointer-button-press` as the *gesture*. The default values for the handlers for all gestures is `nil`, meaning there is no handler. The presentation subsystem could be implemented by installing `input-context-button-press-handler` as the handler for pointer button presses.

Chapter 23

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.

Issue: What exactly is a presentation type? 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*

23.1 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.

Issue: What exactly is a presentation? What does it mean for presentations to be nested? — SWM

⇒ `presentation` [*Protocol Class*]

The protocol class that corresponds to a presentation. A subclass of `output-record`. If you want to create a new class that behaves like a presentation, it should be a subclass of `presentation`. Subclasses of `presentation` must obey the presentation protocol.

⇒ `presentationp` *object* [*Function*]

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

⇒ `standard-presentation` [*Class*]

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

⇒ `:object` [*Init Arg*]

⇒ `:type` [*Init Arg*]

⇒ `:view` [*Init Arg*]

⇒ `:single-box` [*Init Arg*]

⇒ `:modifier` [*Init Arg*]

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.

23.1.1 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)` *new-object presentation* [*Generic Function*]

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

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

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

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

Changes the object associated with the *presentation presentation* to *new-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)` *new-single-box presentation* [*Generic Function*]

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

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

Returns the "modifier" associated with the *presentation presentation*.

23.2 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.

23.2.1 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.

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.

23.2.2 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 in `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.

23.2.3 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:

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.

⇒ `call-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.

If *presentation-function-name* is `eq` to the symbol `apply`, then the *presentation-function-name* argument is set to be the first element of *arguments*, and *arguments* is set to be the remainder of *arguments*. In this case, the presentation generic function is called with `apply` instead of `funcall`.

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

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

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

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 should 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.

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` [*Presentation Method*]

Issue: To be supplied. — SWM

⇒ `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)

23.2.4 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.

⇒ `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. 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.

⇒ `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.

23.3 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 record-type)* `&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*.

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 (:object #p"foo" :type 'pathname)
  (princ "FOO"))
```

⇒ `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 the class of *object* (or one of its superclasses) if that class is a valid presentation type, otherwise it defaults to `expression`. 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*.

23.4 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, this the presentation-based input mechanism supports *context-dependent input*.

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 is the context established by the most recent call to `with-input-context`, and the last element is 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.

⇒ `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*`. 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 prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures blip-gestures additional-blip-gestures* [*Function*]

Requests input of type *type* from the stream *stream*, which defaults to `*query-io*`. `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-internal`.

`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 prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures blip-gestures additional-blip-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-internal`.

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

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-internal` is provided as a convenient function for the `stream-accept` methods to call when they require the default behavior.

⇒ `accept-internal` *stream type* `&key` *view default default-type provide-default insert-default replace-input history prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures blip-gestures additional-blip-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-internal` returns two values, the object representing the input and its presentation type.

`accept-internal` establishes an input context via `with-input-context`, and then calls the `accept` presentation method for *type* and *view*. 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.

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

If the user clicked on a matching presentation, `accept-internal` 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-internal` 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*.

Issue: What is a presentation type history? Should they be exposed? — SWM

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.

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.

blip-gestures is a list of gesture names that will override the current blip gestures (which are stored in `*blip-gestures*`). Alternatively, *additional-blip-gestures* can be supplied to add blip gestures without overriding the current ones. See Chapter 24 for a discussion of blip 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* `&rest` *accept-args* `&key` [*Generic Function*]

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

⇒ `prompt-for-accept-internal` *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`.

23.5 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`. Subclasses of `view` must obey the view protocol.

All of the view classes are immutable.

⇒ `viewp` *object* [*Function*]

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

⇒ `textual-view` [*Class*]

The 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 class that represents the default view that is used inside `menu-choose`. It is a subclass of `textual-view`.

⇒ `textual-dialog-view` [*Class*]

The class that represents the default view that is used inside `accepting-values` dialogs. It is a subclass of `textual-view`.

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

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

⇒ `iconic-view` [*Class*]

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

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

These are objects of class `textual-view`, `textual-menu-view`, `textual-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.

23.6 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.

23.6.1 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`.

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 position *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, and should not be saved 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-click-and-drag-translator` *name from-type to-type destination-type command-table* `&key` *gesture tester documentation pointer-documentation menu priority feedback highlighting* [*Macro*] arglist `&body` body)

Defines a "direct manipulation" translator named *name* that translates from objects of type *from-type* to objects of type *to-type* when the "from presentation" is "dragged" over 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, presses a pointer button matching *gesture*, drags the "from presentation" to a "to presentation" by moving the pointer, and then release the pointer button. When the pointer button is released, the translator whose *destination-type* matches the presentation type of the "to presentation" is chosen.

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 `frame-click-and-drag-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-click-and-drag-highlighting`.

The other arguments to `define-presentation-to-command-translator` are the same as for `define-presentation-translator`.

23.6.2 Presentation Translator Functions

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

Returns a list of all the translators associated with *frame*'s current 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: `find-presentation-translators` will probably 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.

⇒ `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 of translators that definitely apply to the presentation *presentation* in the input context *input-context*. 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.)

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*.

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.

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

23.6.3 Finding Applicable Presentations

⇒ `find-innermost-applicable-presentation` *input-context window x y* `&optional` *modifier-state frame* [*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. *modifier-state* is a mask that describes what modifier keys are held down on the keyboard, and defaults to the values that corresponds to whatever modifier keys are currently held down for *window* (see `event-modifier-key-state`).

frame defaults to `*application-frame*`.

⇒ `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 "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.

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.

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

Unhighlights any highlighted presentations on *stream*.

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

Produces pointer documentation for the presentation *presentation* in the input context *input-context* on the window *window* at the point indicated by *x* and *y*, on behalf of the application frame *frame*. The documentation is written to the stream *stream*.

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

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

23.6.4 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.

Issue: Specify the order of translator search so shadowing can be done rationally. — SWM

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

23.7 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).

23.7.1 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.

23.7.2 Numeric Presentation Types

⇒ `number` [*Presentation Type*]

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

⇒ `complex` [*Presentation Type*]

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

⇒ `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`.

23.7.3 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.

23.7.4 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.

23.7.5 "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`.

23.7.6 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.

23.7.7 "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.

⇒ `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`.

23.7.8 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.

23.7.9 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.

⇒ `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 are handled by the encapsulated interactive stream, but some operations are handled directly by the input editing stream itself. (See Appendix D 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 call `immediate-rescan` when they are done modifying the buffer.

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* [*Function*]

Returns *true* if *object* is an interactive stream, that is, a bidirectional stream intended for user interactions. Otherwise it returns *false*. 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`. Subclasses of `input-editing-stream` must obey the input editing stream protocol.

⇒ `input-editing-stream-p` *object* [*Function*]

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 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*) `&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), `*query-io*` 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; it 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)* `&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*. `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), `*query-io*` 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.

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 array with a fill pointer. The fill pointer of the array 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* [*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*.

⇒ `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 editor 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` should 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 set of input editing commands and key bindings, taken roughly from EMACS.

Input editor command	*Suggested 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.

24.2 Activation and Blip 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.

Blip 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*.

⇒ `*blip-gestures*` [*Variable*]

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

⇒ `with-blip-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 blip gestures. If it is *false* (the default), then *gestures* will be added to the existing set of blip gestures. `with-blip-gestures` must bind `*blip-gestures*` to the new set of blip gestures.

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

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

Returns *true* if the gesture object *gesture* is a blip gesture, otherwise returns *false*.

⇒ `*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.

24.3 Signalling Errors Inside `present` Methods

⇒ `simple-parse-error` [*Condition*]

The error that is signalled by `simple-parse-error`. This is a subclass of `parse-error`.

⇒ `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` [*Condition*]

The error that is signalled by `input-not-of-required-type`. This is a subclass of `parse-error`.

⇒ `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, this should be treated as a single gesture.

⇒ `read-token` *stream* `&key` *input-wait-handler pointer-button-press-handler click-only* [*Function*]

Reads characters from the input stream *stream* until it encounters an activation blip, or pointer gesture. Returns the accumulated string that was delimited by the activation blip gesture, leaving the delimiter unread.

If the first character of typed input is a quotation mark (`#\"`), then `read-token` will ignore blip 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` should 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 stream *stream*. If *acceptably* is *true* and there are any characters in the token that are blip gestures (see the macro `with-blip-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 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 `:possibilites` 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 *interactive stream stream*, completing over a set of possibilities.

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 one 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 an activation gesture or a blip gesture that is not a partial completers.
- `: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 one 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*`).

⇒ `completing-from-suggestions` *(stream* `&key` *partial-completers allow-any-input possibility-printer)* `&body` *body* [*Macro*]

Reads input from the interactive 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), `*query-io*` 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.

⇒ `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

Issue: There is a general issue about how these menus fit in with the menus that might be provided by the underlying toolkit. — SWM

⇒ `menu-choose` *items* `&key` *associated-window printer presentation-type default-item default-style label cache unique-id id-test cache-value cache-test max-width max-height n-rows n-columns x-spacing y-spacing cell-align-x cell-align-y 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`.

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 used if this item is selected.
- `:documentation`—associates some documentation with the menu item. When *:pointer-documentation* is not `nil`, it will be used a pointer documentation for the item.

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.

default-style is a text style that defines how the menu items are presented.

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 be able 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 be able 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.

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 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. It then draws the menu items into that window using the drawing function. The drawing function gets called with arguments (*stream type*). It can use *type* for its own purposes, the usual being using it for present.

menu-choose-from-drawer returns two values: the object the user clicked on, and the 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.

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 *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.

⇒ print-menu-item *menu-item* &optional *stream* [*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

Issue: There is a general issue about how these dialogs fit in with the dialogs that might be provided by the underlying toolkit. — SWM

⇒ `accepting-values` *(*`&optional` *stream* `&key` *own-window exit-boxes initially-select-query-identifier resynchronize-every-pass label x-position y-position 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 "abort" and "done" 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), `*query-io*` 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 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). Then `accepting-values` 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 this 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.

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* be 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.

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"))
```

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.

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.

label is as for `menu-choose`. *x-position* and *y-position* are as for `menu-choose-from-drawer`.

⇒ `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 exit-boxes* [*Generic Function*]

Interprets the *exit-boxes* specification (which is as for `accepting-values`) and displays the exit boxes for the dialog frame *frame* on the stream *stream*. The default method (on `accept-values`) simply writes a line of text associated the Exit and Abort strings with presentations that either exit or abort from the dialog.

The *frame* and *stream* 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. *prompt* may be either a string (which will be displayed via `write-string`), or a form that will be evaluated to draw the button. *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), `*query-io*` is used.

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-item*, where *menu-item* will be an item in the menu of the command table specified by the `:command-table` option. The default is `nil`, meaning that the command will not be available via menu interactions. If *menu-item* is a string, then that string will be used as the menu name. If *menu-item* is `t`, then the menu name will be generated automatically, as described in `add-command-to-command-table`. Otherwise, *menu-item* 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-item`, 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 translator that translates from the argument to a command. 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 menu items (used to support interaction via command menus). The menu items can invoke commands or sub-menus. The menu does not contain any items inherited from superior command tables.

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`. Subclasses of `command-table` must obey the command table protocol. Members of this class are mutable.

⇒ `command-table-p` *object* [*Function*]

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.

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* [*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.)

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 (errorp t)* [*Function*]

Creates a command table named *name*. *inherit-from* and *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` [*Condition*]

The class that is the superclass of the following four conditions. This class is a subclass of `error`.

⇒ `command-table-not-found` [*Condition*]

The error that is signalled by such functions as `find-command-table` when a command table is not found.

⇒ `command-table-already-exists` [*Condition*]

The error that is signalled when the programmer tries to create a command table that already exists.

⇒ `command-not-present` [*Condition*]

The error that is signalled when a command is not present in a command table.

⇒ `command-not-accessible` [*Condition*]

The error that is signalled when a command is not accessible in a command table.

⇒ `command-already-present` [*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. (The automatically generated name consists of the command's name with the hyphens replaced by spaces, and the words capitalized; furthermore, if there is a prefix of "COM-", the prefix is removed. For example, if the command name is `com-show-file`, the command-line name will be "Show File".) For the purposes of command-line name lookup, the character case and style of *name* are ignored.

menu 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 it is `t`, an automatically generate menu name will be used. When it 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.

⇒ `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-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`.

⇒ `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 and style.

⇒ `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.

⇒ `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.The command table used by CLIM to store

⇒ `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 menu items. The menu specifies a mapping from a menu name (the name displayed in the menu) to a 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.

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 menu item, and `:text-style`, which is used to indicate what text style should be used for this 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 and style of the 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 menu item to *command-table*'s menu. *string* is the name of the menu item; its character case and style are ignored. *type* is either `:command`, `:function`, `:menu`, or `:divider`. *command-table* is a command table designator.

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 a a menu item 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 menu item.

text-style is either a text style spec or `nil`. It is used to indicate that the 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 menu 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 and style of *string* are 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 menu 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 and style of *string* are ignored when searching the command table's menu.

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

Applies *function* to all of the menu 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 menu item; it has dynamic extent. The menu items are mapped 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 menu item to see if it is `:menu`.

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

Given a menu name and a command table, returns two values, the 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 the menu item is not present in *command-table* and *errorp* is true, then the `command-not-accessible` error will be signalled. Note that the character case and style of *string* are ignored when searching the command table's menu.

⇒ `command-menu-item-type` *menu-item* [*Function*]

Returns the type of the menu item *menu-item*, for example, `:menu` or `:command`. If *menu-item* is not a menu item, the result is unspecified.

⇒ `command-menu-item-value` *menu-item* [*Function*]

Returns the value of the 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 menu item, the result is unspecified.

⇒ `command-menu-item-options` *menu-item* [*Function*]

Returns a list of the options for the menu item *menu-item*. If *menu-item* is not a 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 (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*, *initial-spacing*, *cell-align-x*, *cell-align-y*, and *move-cursor* are as for `formatting-item-list`.

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

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

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

Issue: Should this be generic on application frames? — SWM

27.4 Keystroke Accelerators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that *gesture* may be either a keyboard gesture name of a gesture object. When it is a gesture name, `eql` will be used to compare the supplied gesture to the gesture names stored in the command table's menu. When it is a gesture object, `event-matches-gesture-name-p` will be used to do the comparison.

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

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

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

Unlike `find-keystroke-item`, this follows the sub-menu chains that can be created with `add-menu-item-to-command-table`. If the keystroke accelerator cannot be found in the command table or any of the command tables from which it inherits, `lookup-keystroke-item` will return `nil`. *command-table* is a command table designator.

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

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

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

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

Issue: The above three functions need to have their hands on the port object since they are might use `event-matches-gesture-name-p`*, which also needs to have a port object. — SWM*

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

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

27.5 Presentation Translator Utilities

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

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

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

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

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

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

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

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

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

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

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

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

27.6 The Command Processor

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

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

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

command-parser must be a function of two arguments, a command table and a stream. It reads a command from the user and returns a command object. The default value for *command-parser* is `*command-parser*`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

⇒ `*command-parser*` [*Variable*]

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

⇒ `*command-unparser*` [*Variable*]

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

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

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

⇒ `*unsupplied-argument-marker*` [*Variable*]

The value of `*unsupplied-argument-marker*` is an object that can be uniquely identified as standing for an unsupplied argument in a command object.

⇒ `*numeric-argument-marker*` [*Variable*]

The value of `*numeric-argument-marker*` is an object that can be uniquely identified as standing for a numeric argument in a command object.

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

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

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

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

27.6.1 Command Presentation Types

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

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

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

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

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

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

The textual representation of a `command-name` object is the command-line name of the command, while the internal representation is the command name.

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

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

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

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

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

Chapter 28

Application Frames

Issue: Integrate the CLIM 0.9 stuff, but try to address the issues that DCPL raises and all of the stuff in the mail as well. — SWM

Issue: The Silica-based macrology for defining panes and layouts doesn't easily allow, if it allows at all, for sharing panes between multiple layouts. CLIM 1.0 seems to have this capability, but the syntax for the constraints is a bit baroque. Perhaps there just isn't a good one, since in all these years we've never found it. Another way to say all of this is that the functional aspects of frames (panes and streams) needs to be separated from the layout description. — DCPL

Issue: We need to decide how much of the pane, stream and layout functionality we want to present in the API and/or SPEC, and how to present it. For example, CLIM 0.9's `frame-pane` *isn't conceptually elegant because that is how you get the current layout, and the layout is a pane, and you have to know that. Better is* `frame-current-layout` *and then have layout-XXX functions to interrogate the layout. I think CLIM 1.0 does better here. — DCPL*

Issue: CLIM 0.9's method of having panes names be slot names is poor because it is implicit, and problematic because it uses `slot-unbound`, *which while it may work, shoots up a red flag. DW and presumably CLIM 1.0 have ways to ask for the pane names and ways to retrieve a pane from a frame given its name. I think this is the right model. The tricky thing, which Silica may allow and that we probably want to preserve if possible, is for a named pane to have another named pane as one of its descendants. Also have a named pane have a named stream as one of its descendants (in the event we have streams that aren't panes). This is important for viewport panes pointing at streams. — DCPL*

Issue: The concept of a frame manager is an important one, as it is the entity that bridges the gap between the application and the host windowing system. — DCPL

Issue: The concept of server paths from CLIM 0.9 is superior to the explicit `open-root-window` *paradigm of CLIM 1.0. CLIM 0.9 has some problems in this regard, but I have some saved mail from an ILA conversation a while that goes a long way toward removing those problems. — DCPL*

28.1 Overview

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

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

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

Some applications may act as frame managers that allow the frames of other applications to be displayed with their own frames. For example, a text editor might allow figures generated by a graphic editor to be edited in place by managing the graphics editor's frame within its own frame.

Application frames provide support for a standard interaction processing loop, like the Lisp "read-eval-print" loop, called a *command loop*. The application programmer has to write only the code that implements the frame-specific commands and output display functions. A key aspect of the command loop is the separation of the specification of the frame's commands from the specification of the end-user interaction style.

The standard interaction loop consists of reading an input "sentence" (the command and all of its operands), executing the command, and updating the displayed information as appropriate. CLIM implementations are free to run the display update part of the loop at a lower priority than command execution, for example, some implementations may choose not to update the display if there is typed-ahead input. Also, command execution and display will not occur simultaneously, so user-defined functions need not have to cope with multiprocessing.

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

- Defines the presentation types that are specific to the application.
- Defines the application frame's commands, specifying the presentation types of the operands

involved in each command.

- Defines the set of frames needed to support the application.
- Defines the output display functions associated with the frames (possibly using other facilities such as the incremental redisplay).

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

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

28.2 Defining and Creating Application Frames

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

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

All application frame classes are mutable.

⇒ `application-frame-p` *object* [*Function*]

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

⇒ `:name` [*Init Arg*]

⇒ `:pretty-name` [*Init Arg*]

⇒ `:panes` [*Init Arg*]

⇒ `:command-table` [*Init Arg*]

⇒ `:disabled-commands` [*Init Arg*]

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

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

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

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

Defines a frame and CLOS class named by the symbol *name* that inherits from *superclasses* and has state variables specified by *slots*. *superclasses* is a list of superclasses that the new class will inherit from (as in `defclass`). When *superclasses* is `nil`, it behaves as though a superclass of `standard-application-frame` was supplied. *slots* is a list of additional slot specifiers, as in `defclass`. Each instance of the frame will have slots as specified by these slot specifiers. These slots will typically hold any per-instance frame state.

options is a list of `defclass`-style options, and can include the usual `defclass` options `defclass`, plus any of the following:

- `:pane` *form*, where *form* specifies the single pane in the application. There is no default for this option. This is the simplest way to define a pane hierarchy. The `:pane` and `:panes` options are mutually exclusive. See below for a complete description of the `:pane` option.
- `:panes` *form*, where *form* specifies the panes of the application. There is no default for this option. The `:pane` and `:panes` options are mutually exclusive. See below for a complete description of the `:panes` option.
- `:layout` *form*, where *form* specifies the layout. The default layout is to lay the panes out in horizontal strips. See below for a complete description of the `:layout` option.
- `:command-table` *name-and-options*, where *name-and-options* is a list consisting of the name of the application frame's command table followed by some keyword-value pairs. The keywords can be *inherit-from* or *menu* (which are as in `define-command-table`). The default is to create a command table with the same name as the application frame.
- `:command-definer` *value*, where *value* either `nil`, `t`, or another symbol. When it is `nil`, no command-defining macro is defined. When it is `t`, a command-defining macro is defined, whose name is of the form `define-`*name*`-command`. When it is another symbol, the symbol names the command-defining macro. The default is `t`.
- `:disabled-commands` *commands*, where *commands* is a list of command names that are initially disabled in the application frame.
- `:command-menu` *command-table* specifies a set of commands that the user can directly invoke from the frame in some frame manager specific way, such as a menu bar on a Macintosh. In other words, it specifies the top-level commands of the application. *command-table* is the command table whose menu will be used in the "menu bar". If defaults to the command table for the application.

- `:top-level` *form*, where *form* is a list whose first element is the name of a function to be called to execute the top-level loop. The function must take at least one argument, the frame. The rest of the list consists of additional arguments to be passed to the function. The default function is `default-frame-top-level`.
- `:resize-frame` *resize-p*, where if *resize-p* is true, this indicates that the frame is willing to be resized when panes in its current hierarchy change their space requirements or when a new hierarchy is selected by changing the frame's current layout.

Issue: Explain what gets evaluated when. — SWM

Issue: What does `:layout` *look like? TeX-like? Genera constraint frames? — SWM*

The value of the `:panes` option is a list of pane descriptions. Each pane descriptions is a cons consisting of a pane name (which is a symbol), a pane type, and a list of keyword-value pairs.

The pane options usable by all panes types are:

- `:default-text-style` *text-style*.
- `:scroll-bars` *scroll-bar-spec*.
- `:default-size` *size-spec*.
- `:display-function` *display-spec*, where *display-spec* is a list whose first element is the name of a function to be applied to the application frame, the pane, and the rest of *display-spec*. Two predefined display functions are `display-command-menu` (in which case a typical display-spec would be `` `(display-command-menu ,``*command-table*`)`), and `incremental-redisplay-display-function` (in which case a typical display-spec would be `'(incremental-redisplay-display-function` *display-function*`)`).
- `:display-time` *time* controls when the pane's displayer function is called. When *time* is `nil`, the pane's displayer is not called. When *time* is `t`, the displayer is called the "next time", and then the display time is set to `nil`. When *time* is `:command-loop`, the displayer is run each time around the application frame's command loop.

When a pane uses incremental redisplay (via `incremental-redisplay-display-function`), CLIM supplies the outermost call to `updating-output`, saves the incremental redisplay record, and calls `redisplay`. The function that does the output performs only the nested calls to `updating-output`. If an application pane uses incremental redisplay without using the `incremental-redisplay-display-function`, it must perform the outermost call to `updating-output`, save the incremental redisplay record, and call `redisplay` itself.

Issue: `:panes` *is wrong, get the real scoop from Dennis or Bill. Their description is that the* `:panes` *option is used to define multiple named top-level panes. Its format is a list of two element lists. The first element of each sublist is a symbol naming the layout. The second element is a form that evaluates to a pane. Using* `setf` *on* `frame-current-layout` *can be used to switch between the defined layouts. The* `generate-pane` *method supplied will also take into account the current frame layout though it will use the first layout as a default. — SWM*

Issue: ILA seems to have flushed pane types altogether and hand all of that stuff to the `:display-function` *and* `:display-time`*? Not a bad idea. On the other hand, it's nice to be able to specify a name for a canned set of pane options instead of specifying the options individually. — SWM*

Issue: What are the defaults for the pane options? — SWM

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

This macro provides lexical access to the "current" frame for use with the `:pane`, `:panes`, and `:layout` options. *frame* is bound to the current frame within the context of one of those options. *body* may have zero or more declarations as its first forms.

⇒ `make-application-frame` *frame-type* `&rest` *options* `&key` *frame-manager enable shrink state left top right bottom width height* `&allow-other-keys` [*Function*]

Makes an instance of the application frame of type *frame-type*. *options* are passed as additional arguments to `make-instance`, after the frame manager and sizing options have been removed.

left, *top*, *right*, *bottom*, *width*, and *height* can be used to specify the size of the frame.

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

Issue: Are there standard options, such as what display to put the window on? — SWM

Issue: What is the documented interface for selecting/exposing the frame? Should there be a way to find-or-create a frame for a given application? Or is that too much into the window-manager field? — SWM

⇒ `*application-frame*` [*Variable*]

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

28.3 The Application Frame Protocol

The generic functions described in this section describe the application frame protocol. All classes that inherit from `application-frame` must inherit or implement methods for all of these functions.

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

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

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

Sets the name of the frame *frame* to *new-name*, which must be a symbol.

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

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

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

Sets the pretty name of the frame *frame* to *new-name*, which must be a string.

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

Returns the command table for the frame *frame*.

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

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

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

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

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

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

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

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

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

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

Issue: How can the above four functions find a meaningful pane given the Silica model of describing frame panes? — SWM

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

This will be bound either to `nil` or to a stream to which pointer documentation should be written.

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

Returns the stream that will be used for `*pointer-documentation-output*` for the *frame frame*. If this returns `nil`, no pointer documentation will be generated for this frame.

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

Returns the sheet that is the top-level sheet for the *frame frame*. This is the sheet that has as descendents all of the panes of *frame*.

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

Returns a list of all of the *frame frame*'s named panes. This includes panes in the current layout, and panes in other layouts as well. This function returns objects that reveal CLIM's internal state; do not modify those objects.

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

Returns a list of those named panes in the *frame frame*'s current layout. This function returns objects that reveal CLIM's internal state; do not modify those objects.

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

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

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

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

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

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

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

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

Lays out all of the panes in the *frame frame* using the sheet *top-level-sheet* as the parent sheet of the frame's panes. The panes are layed out according to the layout specified by `frame-current-layout`.

⇒ `frame-exit` [*Restart*]

The restart that is invoked when `frame-exit` is called.

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

Exits from the *frame frame*. The default method (on `standard-application-frame`) invokes the `frame-exit` restart.

⇒ `panes-need-redisplay` *frame* [*Generic Function*]
⇒ `pane-needs-redisplay` *frame pane* [*Generic Function*]

`panes-need-redisplay` causes all of the panes in the frame *frame* to be redisplayed the next time around the command loop. `pane-needs-redisplay` causes only *pane* within *frame* to be redisplayed the next time around the command loop; in this case, *pane* is either a pane or the name of a named pane.

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

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

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

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

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

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

⇒ `redisplay-frame-command-menu` *frame* [*Generic Function*]

When a command table gets modified (by adding or removing a command, or adding or removing a menu item, or by enabling or disabling a command), the command menus for application frames may need to be redisplayed. `redisplay-frame-command-menu` is responsible for doing this.

⇒ `notify-user` *frame format-string* `&rest` *format-args* [*Generic Function*]

Notifies the user of some event on behalf of the frame *frame*. *format-string* and *format-args* are arguments allowable by `format`.

This function is intended to provide a look and feel independent way for applications to communicate messages to the user. For example, a frame manager might provide a top-level message window or a message window for each frame, or might pop up an alert box.

28.3.1 Interface with Presentation Types

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

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

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

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

Locates and returns the innermost applicable presentation on the window *stream* whose sensitivity region contains the point indicated by *x* and *y*, on behalf of the frame *frame* in the input context *input-context*.

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

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

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

The default implementation (on `standard-application-frame`) unhighlight any highlighted presentations, find the applicable presentation with `frame-find-innermost-applicable-presentation-at-position`, and then call `throw-highlighted-presentation` to execute the translator that corresponds to the user's gesture.

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

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

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

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

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

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

28.3.2 The Generic Command Loop

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

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

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

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

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

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

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

The `:around` method of `run-frame-top-level` on the `application-frame` class is responsible for binding `*application-frame*` to *frame*.

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

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

`default-frame-top-level` will also establish a simple restart for `abort`, and bind the standard stream variables as follows. `*standard-input*` will be bound to the value returned by `frame-standard-input`. `*standard-output*` will be bound to the value returned by `frame-standard-output`. `*query-io*` will be bound to the value returned by `frame-query-io`. `*error-output*` will be bound to the value returned by `frame-error-output`. It is unspecified what `*debug-io*` will be bound to.

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

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

⇒ `read-frame-command` *frame stream* [*Generic Function*]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

28.4 Window Viewport Management

The *viewport* of a window is the area of the window's drawing plane that is currently visible in the window.

Issue: Where does this discussion of viewports really belong? — SWM

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

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

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

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

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

Returns the viewport of *window*, which is an object of type `area`. Usually the viewport will be a `standard-rectangle`.

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

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

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

Returns two values, the x and y position of the top-left corner of *window*'s viewport.

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

Sets the position of *window*'s viewport to *x* and *y*.

28.5 Frame Managers

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

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

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

28.5.1 Finding Frame Managers

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

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

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

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

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

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

⇒ `find-frame-manager &key` *where* [*Function*]

Finds an appropriate frame manager that conforms to the values of a number of parameters including the *where* argument, special dynamic contexts established by the application or CLIM on the application’s behalf, and global variables established by the programmer or user.

The value of the *where* option can be a port or a server path or `nil`, which is its default. This value is used to screen candidate frame managers to ensure that they are associated with the correct port. A frame manager is found using the following rules in the order listed:

1. If a current frame manager has been established via an invocation of `with-frame-manager`, as is the case within a frame's top-level, and that frame manager conforms to *where*, it is returned. Any frame managers conforms if *where* is `nil`.

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

3. If *where* is a port, then a frame manager that conforms to the default look and feel of the port is found or constructed and returned.

4. If *where* is a server path, then a port is found for that server, and then a frame manager that conforms the default look and feel of the port is found or constructed and returned.

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

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

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

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

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

28.5.2 Frame Manager Operations

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

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

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

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

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

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

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

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

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

These functions return the pane of the frame and the frame of the pane, respectively. `pane-frame` can be invoked on any pane in a frame's pane hierarchy, not just its single root pane.

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

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

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

This function is invoked by a standard method of `adopt-frame`. It is the responsibility of the frame implementor to provide a method that invokes `setf` of `frame-pane` on the frame with a value of type `pane`. `define-application-frame` automatically supplies a `generate-panes` method if the `:panes` option is used.

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

Resizes the frame and layouts out the current pane hierarchy according to the layout protocol. The basics of the layout protocols are described in Section 29.3. This function is automatically invoked on a frame when it's adopted, after its pane hierarchy is generated.

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

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

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

⇒ `reconfigure-frame` *frame pane-name* `&key` *resize-frame* [*Generic Function*]

Switches the top-level pane to the pane specified by *pane-name*. This function can only be used in conjunction with the `:panes` options to `define-application-frame`, which is used to name panes.

If *resize-frame* is nil, then the frame is not resized to accommodate the new pane hierarchy.

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

Returns the list of possible top-level pane names. This is the set of valid inputs to the *pane-name* argument of `reconfigure-frame`.

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

Returns the name of the current top-level pane of the frame.

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

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

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

Frame properties can be used to associate frame specific data with frames without adding additional slots to the frame's class. However, for important properties, the frame developer want to add slots to store highly utilized information.

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

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

28.5.3 Frame Manager Settings

The Frame Manager Settings facility provides a mechanism for communicating information from frames to their frame managers in a way that users can control or override decisions.

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

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

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

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

28.6 Examples of Applications

```
;; Define the mapping between the top-level command names and
;; the commands they invoke.
(define-menu-group insdel-menu
  (("Insert Pane" com-insdel-frame-insert-pane)
   ("Remove Pane" com-insdel-frame-remove-pane)))

;; Simple application frame
(define-application-frame insdel-frame ()
  (vbox
    (added-panes :initform nil))
  (:pane
    ;; Get access to vbox slot
    (with-frame-slots (vbox)
      ;; Macro to create a vertical stack of panes.
      (vertically
        (realize-pane 'line-editor-pane
                      :width '(30 :character) :text "This is a line editor")
        (realize-pane 'horizontal-divider :size :large)
        (restraining
          (setf vbox
                (vertically
                  (realize-pane 'label-pane
                                :spacing 3 :label "This is a label pane")
                  (realize-pane 'horizontal-divider :size :medium)
                  :fill))))))
  (:menu-group insdel-menu))

;; The :COMMAND-DEFINER T option implied by use of :MENU-GROUP
;; caused the following macro to be defined.  It is basically like
;; DEFINE-COMMAND, but automatically associates the command with this
;; particular application.
(define-insdel-frame-command com-insdel-frame-insert-pane ()
  (with-frame-slots (added-pane vbox)
    ;; This command creates a new label pane (and divider) and pushes
    ;; it onto the panes list.  Each invocation adds one more pane
    ;; Inserting after :fill.
    (push (realize-pane 'hline-pane :size :small)
          added-panes)
    (push (realize-pane 'label-pane
                        :label "Another ...")
          added-panes)
    ;; Not exported from CLIM yet
    (ws:insert-panes vbox
                     (list
                       (cadr added-panes)
                       (car added-panes)))
    (repaint-sheet vbox (sheet-region vbox))
```

```
    (notify-user (pane-frame vbox) "Pane Inserted")))

;; Remove one of the previously-added panes.
(define-insdel-frame-command com-insdel-frame-remove-pane ()
  (declare (ignore button))
  (with-slots (added-pane vbox) frame
    (if added-panes
        (progn
          (ws:remove-pane vbox (pop added-panes))
          (ws:remove-pane vbox (pop added-panes))
          (repaint-sheet vbox (sheet-region vbox))
          (notify-user (pane-frame vbox) "Pane removed"))
        (notify-user (pane-frame vbox) "Trying to delete my panes, eh?"))))

(defun insdel (&rest settings)
  (make-frame 'insdel-frame
              :title "Insdel" :width 400 :height 200
              :user-settings settings
              :frame-manager (find-frame-manager)
              :enable t))
```

Chapter 29

Panes

CLIM panes are similar to the gadgets or widgets of other toolkits. They can be used by application builders to compose the top-level user interface of their applications as well as auxiliary dialogues. The application programmer provides an abstract specification of the pane hierarchy, which CLIM uses in conjunction with user preferences and other factors to select a specific "look and feel" for the application. In some environments a CLIM application can use the facilities of the host window system toolkit via a set of *adaptive panes*, allowing a portable CLIM application to take on the look and feel of a native application user interface.

Panes are rectangular objects that are implemented as special sheet types. An application will typically construct a tree of panes that will divide up the screen space allocated to the application. The various CLIM pane types can be characterized by whether they have children panes or not: panes that can have other panes as children are called *composite panes*, and those that don't are called *leaf panes*. Composite panes are used to provide a mechanism for spatially organizing ("laying out") other panes. Leaf panes implement gadgets that have some appearance and react to user input by invoking application code. Another kind of leaf pane provides an area of the application's screen real estate that can be used by the application to present application specific information. CLIM provides a number of these *application pane* types that allow the application to use CLIM's graphics, input, stream, and extended stream facilities.

29.1 Abstract Panes

Abstract panes are panes that are defined only in terms of their programmer interface, or behavior. The protocol for an abstract pane (that is, the specified set of initialization options, accessors and callbacks) is designed to specify the pane in terms of its overall purpose, rather then in terms of its specific appearance or particular interactive details. The purpose of this abstract definition is to allow multiple implementations of the abstract pane, each defining its own specific look and feel. CLIM can then select the appropriate pane implementation based on factors outside the control of the application, such as user preferences or the look and feel of the host operating environment. A subset of the abstract panes, the *adaptive panes*, have been defined to integrate well across all CLIM operating platforms.

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

29.2 Pane Construction and Uses

Applications typically define the hierarchy of panes used in their frames using the `:pane` or `:panes` options of `define-application-frame`. These options generate the body of methods on functions that are invoked when the frame is being adopted into a particular frame manager, so the frame manager can select the specific implementations of the abstract panes.

There are two basic interfaces to constructing a pane: `make-instance` of a known pane type, and `realize-pane` of an abstract pane protocol name. The `make-instance` approach requires the application to know the name of the specific pane implementation class that is desired; all pane class names, including those of the implementations of abstract pane protocol specifications, end in "-PANE". Using `make-instance` directly circumvents the abstract pane selection mechanism, which is appropriate for some cases. For example, the programmer may want to use particular layout panes like `vbox-pane` and `hbox-pane` and avoid the costs and uncertainty of going through the realization mechanism.

The second approach, `realize-pane`, invokes the "look and feel" realization process to select and construct a pane. Normally this process is implemented by the frame manager, but other realizers are possible. The `realize-pane` function is typically invoked using an abstract pane name, which is always a symbol in the CLIM package that doesn't include the "-PANE" suffix. This naming convention distinguish the names of the abstract pane protocols from the names of classes that implement them. Programmers, however, are allowed to pass any pane class name to `realize-pane`, in which case the frame manager will in general instantiate that specific class.

Issue: We surely want a `pane` *protocol class, which is probably a subclass of* `sheet`. *The instantiable pane classes to which you can do output should also be a subclass of* `medium`. *— SWM*

⇒ `realize-pane` *abstract-pane-protocol-name* `&rest` *initargs* [*Function*]

Selects a class that implements abstract pane protocol specified by *abstract-pane-protocol-name* and constructs a pane of that class. It must be used within the dynamic scope of either a `with-look-and-feel-realization` or the `:pane` or `:panes` options of a `define-application-frame` (which sets up an implicit `with-look-and-feel-realization`).

⇒ `realize-pane-internal` *realizer frame abstract-name* `&rest` *initargs* [*Generic Function*]

This is the generic function that is actually invoked by a call to `realize-pane`. The extant *realizer* is established by `with-look-and-feel-realization`. Frame managers (and other realizers) define a method on this generic function to implement their realization policy.

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

Establishes a dynamic context that installs *realizer* as the entity responsible for realizing panes. All calls to `realize-pane` within this context result in `realize-pane-internal` being invoked on *realizer*. This macro can be nested dynamically; inner uses shadow outer uses. *body* may have zero or more declarations as its first forms.

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

29.2.1 Pane Initialization Options

Client code should be particularly cautious about passing values for these pane options, since the desktop's look and feel or the user's preferences should usually determine these values.

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

These options specify the default foreground and background inks for a pane.

⇒ `:text-style` [*Option*]

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

29.2.2 Pane Queries

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

This function returns the frame that "owns" the pane. `pane-frame` can be invoked on any pane in a frame's pane hierarchy, not just its single root pane, but it can only be invoked on "active" panes, those that are currently adopted into the frame's pane hierarchy.

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

Return the current foreground and background inks of the pane, respectively.

Issue: Why aren't these `medium-foreground` *and medium-background, since panes obey the medium protocol to this extent? — SWM*

29.3 Layout and Layout Panes

In this section, the aspects of the CLIM pane layout protocol that most application builder need to understand are described. A more detailed account of this protocol is presented in Section 31.2.1.

Programmers who need to build their own pane types, particularly, composite pane types, will probably need the more detailed account.

29.3.1 The Basics of the Layout Protocol

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

CLIM uses a two pass algorithm to lay out a pane hierarchy. In the first pass, called space composition, the top-level pane is asked how much space it requires. This may in turn lead to the question being asked recursively of all the panes in the hierarchy, with the answers being composed to produce the top-level pane's answer. Each pane answers the query by returning a *space requirement* (or `space-requirement`) object, which specifies the pane's desired width and height as well as its willingness to shrink or grow along the width and height axis.

In the second pass, called space allocation, the frame manager attempts to obtain the required amount of space from the host window system (of course, this may not be possible). The top-level pane is allocated the space that is actually available. Each pane, in turn, allocates space recursively to each of its descendants in the hierarchy according to the pane's rules of composition.

For many types of pane, the application programmer can indicate the space requirements of the pane at creation time using initialization arguments, as well as, dynamically using the `change-space-requirement` function described below. For example, application panes are used to display application-specific information, so the application can determine how much space should normally be given to them.

Other pane types automatically calculate how much space they need based on the information they need to display. For example, label panes, can automatically calculate their space requirement based on the text they need to display.

A composite pane calculates its space requirement based on the requirements of its children and its own particular rule for arranging them. For example, a pane that arranges its children in a vertical stack would return as its desired height the sum of the heights of its children. Note, however, a composite is not required by the layout protocol to respect the space requests of its children; in fact, composite panes aren't even required to ask their children.

Space requirements are expressed for each of the two dimensions as a preferred size, a mininum size below which the pane cannot be shrunk, and a maxium size above which the pane cannot be grown. (The minimum and maximum sizes can also be specified as relative amounts.) All sizes are specified as a real number indicating the number of device units (such as pixels).

The following pane initialization options control the space request parameters for the pane:

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

These options control space layout. The `:width` and `:height` options specify the preferred horizontal and vertical sizes. The `:max-width` and `:max-height` options specify the maximum amount of space that may be consumed by the pane, and give CLIM's pane layout engine permission to grow the pane beyond the preferred size. The `:min-width` and `:min-height` options specify the minimum amount of space that may be consumed by the pane, and give CLIM's pane layout engine permission to shrink the pane below the preferred size.

If either of the `:max-width` or `:min-width` options is not supplied, it defaults to `:width`. If either of the `:max-height` or `:min-height` options is not supplied, it defaults to `:height`.

`:max-width`, `:min-width`, `:max-height`, and `:min-height` can also be specified as a relative size by supplying a list of the form `(`*number* `:relative)`. In this case, the number indicates the number of device units that the pane is willing to stretch or shrink.

The values of these options are specified in the same way as the `:x-spacing` and `:y-spacing` options to `formatting-table`. (Note that `:character` and `:line` may only be used on those panes that display text, such as a `message-pane`.)

⇒ `+fill+` [*Constant*]

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

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

This function can be invoked to indicate that the space requirements for *pane* have changed. Any of the options that applied to the pane at creation time can be passed into this function as well.

resize-frame determines whether the frame should be resized to accommodate the new space requirement of the hierarchy. If *resize-frame* is true, then `layout-frame` will be invoked on the frame. If *resize-frame* is false, then the frame may or may not get resized depending on the pane hierarchy and the `:resize-frame` option that was supplied to `define-application-framework`.

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

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

Issue: There is still work to do in defining what this means exactly. — ILA

Issue: The protocol for handling space allocation failures (such as a layout that cannot fit in the available space) needs to be specified. The working proposal is that an error will be signalled when the constraints can't be satisfied. If the application chooses, it can establish a condition handler. Otherwise the panes will fall where they may. The `define-frame-application` *macro should provide an option that allows programmers to conveniently specify a condition handler. — ILA*

29.3.2 Layout Panes

Issue: We need to achieve some consensus on what the exact set of layout panes should be and what they should be called. — ILA

The layout panes describe in this section are all composite panes that are responsible for positioning their children according to various layout rules. All of these panes accept the initialization argument `:contents`, the value of which is a list of the children to be laid out.

Layout panes can be selected in the same way as other panes using `make-instance` or `realize-pane`. For convenience and readability of application pane layouts, some of these panes provides a macro that expands into a `make-instance` form, passing a list of the panes created in the body of the macro as the `:contents` argument. For example, you can express a layout of a vertical column of two label panes either as:

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

or

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

Issue: I think that we should flush layout panes from the spec, and leave only the layout macros. Legislating panes overly constrains the implementation, since they are not really necessary to proper laying-out of a frame. — SWM

⇒ `hbox-pane` [*Layout Pane*]
⇒ `horizontally` *(*`&rest` *options)* `&body` *contents* [*Macro*]
⇒ `vbox-pane` [*Layout Pane*]
⇒ `vertically` *(*`&rest` *options)* `&body` *contents* [*Macro*]

These panes horizontally and vertically organize their children, which are the panes created by *contents*.

`hbox-pane` allocates space to its contents from left to right. `vbox-pane` allocates space to its contents from top to bottom. *contents* is list that contains any the following elements:

- A pane. The pane is inserted at this point and its space requirements are used to compute the size.
- A number. The specified number of pixels should be allocated.
- The symbol `+fill+`. This means that an arbitrary amount of space can be absorbed at this point in the layout.

- A list whose first element is a number and whose second element evaluates to a pane. If the number is below 1 then it means that that percentage of excess space or deficit should be allocated to the pane. If the number is greater than 1, then that much space is allocated to the pane. For example:

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

  would create a horizontal stack of two button panes.

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

Similar to the box panes, except that they ensure that all children are the same size in the minor dimension. In other words, these panes are used to create stacks of same-sized items, such as menu items.

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

This pane allows laying panes out in a two-dimensional table arrangement. Each row in *contents* is itself specified as a list of panes.

⇒ `grid-pane` [*Layout Pane*]

A table where each of the cells are the same size in each of the two dimensions.

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

This layout pane reserves some margin space around a single child pane. The space requirement keys that are passed in indicate the requirements for the surrounding space, not including the requirements of the child.

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

This layout pane puts a border around its contents.

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

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

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

⇒ `bboard-pane` [*Layout Pane*]

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

29.3.3 Layout Pane Options

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

The `:align-x` option is one of `:right`, `:center`, or `:left`. The `:align-y` option is one of `:top`, `:center`, or `:bottom`. These have the same semantics as for `formatting-cell`.

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

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

29.4 Application Panes

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

This section describes the existing application pane types. In addition, an application programmer can customize pane behavior by defining a subclass of one of these pane types and writing methods to change the repaint or event-handling behavior.

⇒ `basic-clim-pane` [*Service Pane*]

This pane implements the CLIM graphics protocol and basic event-level protocol. These protocols are documented in earlier chapters of the spec. An application can use this pane type as a "canvas" on which to draw application output.

⇒ `basic-stream-pane` [*Service Pane*]

This pane implements the basic Common Lisp stream protocol for the frame. That is, it implements the basic character stream protocol as described in the Gray proposal to X3J13.

⇒ `extended-stream-pane` [*Service Pane*]

This pane implements the CLIM extended stream protocol. This includes the basic stream protocol plus the extended input and output protocols that support output recording and presentations.

29.4.1 Making Application Panes

An application frequently uses application panes to display more information than will fit in the space allocated to the pane. CLIM provides panes that can be used to add scrolling behavior to the application pane. Since this is a rather common need, a standard macro is provided to package this functionality.

⇒ `making-application-pane` *(*`&rest` *options)* `&body` *contents* [*Macro*]

This macro embeds the contents pane in a collection of composite and leaf panes designed to implement scrolling behavior. These panes can include a viewport pane, which provides a "window" into the application pane's contents and manages the coordinate transformation that determines which part of the contents are visible, two scroll bar gadget panes to provide a user interface to scrolling, and a overseer pane to manage it all. The options are:

- `:scroll-bars`—can be `nil` (no scroll bars), `:horizontal`, `:vertical`, `:both` or `:auto`, which will include scroll bars if the contents pane has more information to display than the viewport can hold.
- `:label`—can be `nil` or a string that will be displayed as a label for the application pane.
- `:text-style`—controls the text style of the label.

Also, the various space-request options (such as `:width` and `:height`) are accepted, controlling the layout of the viewport pane.

29.4.2 CLIM Stream Pane Options

The following are init options to CLIM stream panes.

- `:display-function` specifies a function for CLIM to call to display the contents of a CLIM stream pane. The standard `clim-top-level` top-level function will invoke the pane's display function at the appropriate time (see the `:display-after-commands` option). The value of this option should be either the name of a function to invoke or a cons whose car is the name of a function and whose cdr is additional arguments to the function. The function will be invoked on the frame, the pane, and the additional function arguments, if any.
- `:display-time` tells CLIM when to run the display function. If it is `t`, CLIM will erase the pane's contents and run the display function the next time a frame command is executed. If it is `nil`, CLIM will only run the display function when it is explicitly necessary (when the pane is initially exposed, for example). If it is `:command-loop` the display will be run after any command is executed.

29.5 Pane Options

Panes use a uniform set of initargs to specify their layout and appearance parameters.

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

The `:align-x` option is one of `:right`, `:center`, or `:left`. The `:align-y` option is one of `:top`, `:center`, or `:bottom`. In addition, `:align-y` may be `:baseline` for panes that place text. These have the same semantics as for `formatting-cell`.

⇒ `:trigger` [*Option*]
⇒ `:wire` [*Option*]
⇒ `:value` [*Option*]

These options define how an abstract pane notifies the application frame (or some other pane that is using it to build a more complex pane) that some action (for example, a button has been selected) has occurred. Wires and triggers are describe more completely below. Currently, a client should specify only one of wire (an object that is described below) or trigger (a funcallable object). In addition, value is an object that is passed back to the client via the trigger or the wire according to the definition of the abstract pane.

29.6 Abstract Panes

CLIM defines the following additional abstract panes, each having at least one implementation.

Issue: The full behavior of each pane will be defined after the set of defined panes is finalized. — ILA

⇒ `scrollable-pane` [*Abstract Gadget*]
⇒ `scrolling` *(*`&rest` *options)* `&body` *contents* [*Macro*]

⇒ `one-of-pane` [*Abstract Gadget*]

⇒ `list-pane` [*Abstract Gadget*]

⇒ `text-stream-pane` [*Abstract Gadget*]

⇒ `label-pane` [*Abstract Gadget*]
⇒ `labelled` *(*`&rest` *options)* `&body` *contents* [*Macro*]

⇒ `line-editor-pane` [*Abstract Gadget*]

⇒ `label-button-pane` [*Abstract Gadget*]

⇒ `radio-button-pane` [*Abstract Gadget*]

⇒ `horizontal-divider` [*Layout Pane*]

⇒ `vertical-divider` [*Layout Pane*]

Chapter 30

Gadget Panes

30.1 Overview

Gadget panes are panes that implement such common toolkit components as push buttons or scroll bars. Each gadget pane class has a set of associated generic functions that serve the role of "callbacks" in "traditional" toolkits. For example, a push-button has a "value change" callback function that is invoked when its button is "pressed".

Issue: Doing gadgets as panes seems dangerous, since I thought panes are statically laid out but one should be able to draw a knob anywhere in the drawing plane and have it immediately start working. I would have thought of gadgets as presentations rather than panes. — Moon

The gadget pane definitions supplied here are abstract. That is, the gadget definition does not specify the specific user interface of the gadget pane. In other words, it is not defined whether the user clicks on the button with the mouse or moves the mouse over the button and presses some key on the keyboard to invoke the "value change" callback. Particular toolkit implementations will specify the "look and feel" of their gadgets.

Each of the abstract gadget panes has at least one implementation, written in Lisp using the CLIM input/output kernel capabilities. Its appearance is achieved via calls to the CLIM graphics functions, its interactive behavior defined in terms of the CLIM input event processing mechanism. Since these gadget implementations are written entirely in terms of CLIM, they are portable across all supported CLIM host window systems. However, since the specific look and feel of each such gadget is "fixed" in CLIM Lisp code, the gadget implementation will look and behave the same in all environments.

30.2 Interacting with Gadget Panes

Every gadget has a *client*. The client is specified at gadget creation. The client is made aware of all important user interactions with the gadget via the callback mechanism. Each callback generic function is invoked on both the gadget pane and its client. Typically, a gadget pane's client will be an application frame or a composite pane.

As an example, the argument list for `value-change-callback` looks like *(gadget-pane client gadget-id value)*. Assuming that programmer had defined an application frame called `button-test` that had a `clim-stream-pane` in its slot `output-pane`, he could write the following method:

```
(defmethod value-change-callback
          ((button push-button) (client button-test) gadget-id value)
  (declare (ignore value))
  (with-slots (output-pane) client
    (format output-pane "A button was pressed.")))
```

One problem with this particular example is that it differentiates on the gadget class, not the gadget instance. That is, the same method will run for every push-button that has this frame as its client. To help sort out the various gadgets, each gadget pane has a *gadget id* that can be initialized at gadget creation. The value of this gadget id is passed as the third argument to each callback generic function. In this case, if we had two buttons, we might have installed `start` and `stop` as the respective gadget ids. We could then refine the above as:

```
(defmethod value-change-callback
          ((button push-button) (client button-test) (gadget-id (eql 'start)) value)
  (declare (ignore value))
  (start-test client))

(defmethod value-change-callback
          ((button push-button) (client button-test) (gadget-id (eql 'stop)) value)
  (declare (ignore value))
  (stop-test client))
```

This scheme takes advantage of the multi-method facility in CLOS and frees the programmer from having to subclass each gadget type before using it.

30.3 Abstract Gadgets

The class `push-button` we introduced above is an *abstract gadget class*. The callback interface to all implementations of the gadget is defined by the abstract class. In a frame pane layout specification, a gadget pane is named by its abstract class name. At pane realization time, this abstract class must be resolved to a specific implementation class. The individual implementation classes specify the detailed look and feel of the gadget. Every implementation of an abstract gadget class is a subclass of the abstract class; therefore they all share the same programmer interface. The selection of a specific implementation of an abstract gadget is handled by the frame manager.

30.3.1 Implementation Strategy

```
(defclass gadget-pane (pane)
  ((client :initarg :client :accessor gadget-client)
   (id :initarg :id gadget-id))
  (:default-initargs :id nil :client nil))

;; Abstract definition.
(defclass abstract-push-button () () )

;; For each callback, do this.
(defmethod value-change-callback
           ((gadget abstract-push-button) client gadget-id value)
  (declare (ignore client gadget-id value))
  nil)

;; Implementation side.
;; Associates implementation name with abstract name (somehow)
;; so that frame managers can select it.
(define-abstract-gadget-implementation motif-push-button-pane
                                       (abstract-push-button . supers)
  (slots)
  . option-forms)

--->

(defclass motif-push-button-pane
          (abstract-push-button ,@supers gadget-pane)
  ,slots)
```

As part of pane realization, a frame manager must select (and realize) a specific implementation class for an abstract class. If the abstract name isn't "registered", the abstract name is typically assumed to be the name of a specific (portable) implementation. If the abstract name is registered, the frame manager selects one of the associated implementations. This behavior is implemented by a default method on `realize-frame-pane` for the base `frame-manager` class. Of course, any individual frame manager is free to violate these conventions as part of establishing its peculiar look and feel. It's unwise to shadow completely this default fallback method on `(pane-type t)`.

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

Returns the id of the gadget. The id is typically a simple Lisp object that uniquely identifies the gadgets.

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

Returns the client of the gadget. The client is usually an application frame.

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

Returns the value of this gadget. The interpretation of the value is gadget-specific. For example, a scroll bar's value might be a number between 0 and 1, while a toggle button's value is either `t` or `nil`. See the documentation of individual gadgets to find out how to interpret the value.

⇒ `(setf gadget-value)` *value gadget* `&key` *(call-callbacks* `t`*)* [*Generic Function*]

Sets the gadget's value to the specified value. In addition, if *call-callbacks* is *true*, all the generic operations that would be invoked if the gadget had been clicked upon in normal use will be run. Note that the syntax is:

```
(setf (gadget-value gadget :call-callbacks nil) new-value)
```

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

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

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

Causes the host gadget to become inactive, that is, unavailable for input. In some environments this may cause the gadget to become grayed over; in others, no visual effect may be detected.

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

This is invoked after a gadget is made active (available for input).

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

This is invoked after a gadget is made inactive (unavailable for input).

⇒ `value-change-callback` *gadget-pane client gadget-id* `&optional` *value* [*Generic Function*]

This is invoked when the value of a gadget is changed, either by the user or programmatically. For gadgets that do not have a value, the value of the *value* argument is unspecified and should be ignored.

Programmers will typically class-specialize the *gadget-pane* and *client* arguments, and `eql`-specialize the *gadget-id* argument.

CLIM implementations must implement a default method for `value-change-callback` for every gadget class.

Most gadget classes have a set of additional callbacks, which are described in detail in the description of the gadget class.

30.4 Adaptive Gadget Panes

30.4.1 Overview

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

Each gadget maps to an implementation-specific object that is managed by the underlying toolkit. For example, while a CLIM program manipulates an object of class `scroll-bar`, the underlying implementation-specific object might be an Xt widget of type `Xm_Scroll_Bar`. As events are processed on the underlying object the corresponding generic operations are applied to the lisp gadget.

Refer to `realize-pane` to see how to construct and lay out a gadget.

Issue: Do we want to define something like (adaptive-)gadget-handle? A documented way to get ahold of the underlying toolkit object? — ILA

30.4.2 Predefined Gadgets

CLIM provides a standard set of adaptive gadgets. This section documents the predefined gadget types. Note that not all operations will necessarily be generated by particular toolkit implementations. For example, a user interface toolkit that is designed for a 3-button mouse may generate significantly more gadget events than one designed for a 1-button mouse. More on this below.

push-button Gadget

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

Gadget Init Arguments

`:label`—string

Specifies the label associated with the push button.

`:label-text-style`—text-style

Specifies the text style that will be used when drawing the label.

`:show-as-default-p`—boolean

When *show-as-default-p* is *true*, the push button will be draw with a heavy border, which indicates that this button is the "default operation".

Gadget State Manipulation Operations

⇒ `button-label` *button* [*Generic Function*]

Returns the label of the button.

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

Changes the label of *button* to the new label string *label*.

⇒ `button-text-style` *button* [*Generic Function*]

Returns the text style of the button's label.

⇒ `(setf button-text-style)` *text-style button* [*Generic Function*]

Changes the text style of *button*'s label to the new text style *text-style*.

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

Returns *true* if the push button will be shown as the default, otherwise returns *false*.

⇒ `(setf push-button-show-as-default-p)` *state push-button* [*Generic Function*]

Changes the "show as default" state of *push-button* to *state*, which must be `t` or `nil`.

Gadget Operations

⇒ `value-change-callback` *button client gadget-id value* [*Generic Function*]

Called when the toolkit determines that the button has been pushed. The value of *value* is indeterminate for push buttons, and should be ignored.

⇒ `arm-callback` *push-button client gadget-id* [*Generic Function*]

Called when the pointer is clicked on the push button gadget identified by *gadget-id*, preparatory to a value change. (The exact time when this is called may vary slightly from one toolkit to another.)

⇒ `disarm-callback` *push-button client gadget-id* [*Generic Function*]

Called when the pointer button is released after an `arm-callback`. (The exact time when this is called may vary slightly from one toolkit to another.)

Gadget Comments

The normal sequence of callbacks is `arm-callback`, then `value-change-callback`, then `disarm-callback`.

toggle-button Gadget

The toggle button gadget provides “on/off” switch behavior. This gadget typically appears as a box that is optionally highlighted with a check-mark. If the check-mark is present, the gadget’s value is `t`, otherwise it is `nil`.

Gadget Init Arguments

`:label`—string

Specifies the label associated with the toggle button.

`:label-text-style`—text-style

Specifies the text style that will be used when drawing the label.

`:indicator-type`—`:one-of` or `:some-of`

The indicator-type controls the appearance of the toggle button. For example, many toolkits present a one-of-many choice differently from a some-of-many choice.

Gadget State Manipulation Operations

⇒ `button-label` *button* [*Generic Function*]

Returns the label of the button.

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

Changes the label of *button* to the new label string *label*.

⇒ `button-text-style` *button* [*Generic Function*]

Returns the text style of the button’s label.

⇒ `(setf button-text-style)` *text-style button* [*Generic Function*]

Changes the text style of *button*’s label to the new text style *text-style*.

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

Returns the indicator type of *toggle-button*.

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

Changes the indicator type of *toggle-button* to *type*, which must be `:one-of` or `:some-of`.

Gadget Operations

⇒ `value-change-callback` *toggle-button client gadget-id value* [*Generic Function*]

Called when the toolkit determines that the toggle button has been toggled. The value of *value* is indeterminate for toggle buttons, and should be ignored.

⇒ `arm-callback` *toggle-button client gadget-id* [*Generic Function*]

Called when the pointer is clicked on the toggle button gadget identified by *gadget-id*, preparatory to a value change. (The exact time when this is called may vary slightly from one toolkit to another.)

⇒ `disarm-callback` *toggle-button client gadget-id* [*Generic Function*]

Called when the pointer button is released after an `arm-callback`. (The exact time when this is called may vary slightly from one toolkit to another.)

Gadget Comments

The normal sequence of callbacks is `arm-callback`, then `value-change-callback`, then `disarm-callback`.

scroll-bar Gadget

Gadget Init Arguments

`:orientation`—`:horizontal` or `:vertical`

Specifies the orientation of the scroll bar, either horizontal or vertical.

`:min-value`—real number, default 0

Specifies the minimum value of the scroll bar.

`:max-value`—real number, default 1

Specifies the maximum value of the scroll bar.

`:unit-increment`—real number, default 1

Specifies the increment between tick marks on the scroll bar.

`:page-increment`—real number, default 1

Gadget State Manipulation Operations

⇒ `scroll-bar-orientation` *scroll-bar* [*Generic Function*]

Returns the orientation of the scroll bar.

⇒ `(setf scroll-bar-orientation)` *orientation scroll-bar* [*Generic Function*]

Changes the orientation of the scroll bar to *orientation*.

⇒ `scroll-bar-min-value` *scroll-bar* [*Generic Function*]

Returns the minimum value of the scroll bar.

⇒ `(setf scroll-bar-min-value)` *value scroll-bar* [*Generic Function*]

Changes the minimum value of the scroll bar to *value*.

⇒ `scroll-bar-max-value` *scroll-bar* [*Generic Function*]

Returns the minimum value of the scroll bar.

⇒ `(setf scroll-bar-max-value)` *scroll-bar* [*Generic Function*]

Changes the maximum value of the scroll bar to *value*.

⇒ `scroll-bar-unit-increment` *scroll-bar* [*Generic Function*]

⇒ `(setf scroll-bar-unit-increment)` *scroll-bar* [*Generic Function*]

⇒ `scroll-bar-page-increment` *scroll-bar* [*Generic Function*]

⇒ `(setf scroll-bar-page-increment)` *scroll-bar* [*Generic Function*]

Gadget Operations

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

Invoked each time the "thumb" of the scroll bar is moved while dragging the pointer.

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

Invoked when a pointer button is pressed a some point in the scroll bar.

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

⇒ `scroll-to-top-callback` *scroll-bar client gadget-id* [*Generic Function*]
⇒ `scroll-to-bottom-callback` *scroll-bar client gadget-id* [*Generic Function*]

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

Gadget Comments

`value-change-callback` is invoked when the indicator is released after dragging it.

slider Gadget

Gadget Init Arguments

`:orientation`—`:horizontal` or `:vertical`

`:min-value`—number, default 0

`:max-value`—number, default 1

`:show-value-p`—boolean, default NIL.

`:show-value-p` controls whether or not a numeric display corresponding to the slider's value is visible.

Gadget State Manipulation Operations

⇒ `slider-orientation` *slider* [*Generic Function*]

Returns the orientation of the slider.

⇒ `(setf slider-orientation)` *orientation slider* [*Generic Function*]

Changes the orientation of the slider to *orientation*.

⇒ `slider-min-value` *slider* [*Generic Function*]

Returns the minimum value of the slider.

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

Changes the minimum value of the slider to *value*.

⇒ `slider-max-value` *slider* [*Generic Function*]

Returns the minimum value of the slider.

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

Changes the maximum value of the slider to *value*.

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

⇒ `(setf slider-show-value-p)` *slider* [*Generic Function*]

Gadget Operations

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

Invoked each time the "thumb" of the slider is moved while dragging the pointer.

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

Invoked when a pointer button is pressed a some point in the slider.

Gadget Comments

`value-change-callback` is invoked when the indicator is released after dragging it.

radio-box Gadget

Gadget Init Arguments

`:current-selection`

Gadget State Manipulation Operations

⇒ `radio-box-current-selection` *gadget* [*Generic Function*]

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

Gadget Operations

⇒ `value-change-callback` *radio-box client gadget-id value* [*Generic Function*]

value is the newly-selected button gadget.

Gadget Comments

The radio box enforces selection behavior like car radio buttons. Only one of the buttons managed by the radio box may be “on” at any time. As the current selection changes, the previously selected button and the newly selected button both have their `value-change-callback` handlers invoked. The radio box gadget can manage any other gadget whose gadget-value is constrained to be a boolean. At present, only the toggle button satisfies this criterion. `radio-box-current-selection` is the programmer interface to determining which of the gadgets is selected.

Note that the radio box is the client of its buttons.

cascade-button Gadget

Gadget Init Arguments

`:menu-group`—a menu group

Gadget State Manipulation Operations

⇒ `cascade-button-menu-group` *gadget* [*Generic Function*]

⇒ `(setf cascade-button-menu-group)` *gadget* [*Generic Function*]

Gadget Operations

None

Gadget Comments

When a cascade button is clicked upon, a menu containing its menu group.

Note that the specific user interface to cascade buttons will differ in each underlying toolkit. For example, in some toolkits, merely dragging the mouse across a cascade button is enough to cause its submenu to become visible. In other implementations, the mouse may have to leave the cascade button through a particular boundary. In others, a click may be necessary.

Chapter 31

Defining New Pane Types

31.1 Defining Leaf Panes

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

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

```
;; A new kind of button
(defclass sample-button-pane (gadget-pane) ())

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

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

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

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

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

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

;; This event processing method defines the rather odd interaction
;; style of this button, to whit: it triggers the activate callback
;; whenever the mouse moves into it.
(defmethod enter-region ((pane sample-button-pane) &key &allow-other-keys)
  (value-change-callback pane (gadget-client pane) (gadget-id pane) nil))
```

31.2 Defining Composite Panes

31.2.1 Details of the Layout Protocol

The basics of layout protocol as needed by most application developers are described in Section 29.3. That section should be read first. This details of the layout protocol describe in this section are mostly of interest to implementors of composite panes.

During the space composition pass, a composite pane will typically ask each of its children how much space it requires using `compose-space`. They answer by returning `space-requirement` objects. The composite will then form its own space requirement by composing the space requirements of its children according to its own rules for laying out its children.

During the space allocation pass, a composite pane will arrange its children within the available space and allocate space to them according to their space requirements and its composition rules.

⇒ `compose-space` *pane* [*Generic Function*]

This function is called on a pane to ask for its space requirements. Methods for this function must return a space requirement object.

⇒ `allocate-space` *pane width height* [*Generic Function*]

This function is called on a pane to inform it that it has been allocated space that is *width* by *height*.

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

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

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

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

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

⇒ `space-requirement-width` *space-req* [*Function*]
⇒ `(setf space-requirement-width)` *size space-req* [*Function*]
⇒ `space-requirement-max-width` *space-req* [*Function*]
⇒ `(setf space-requirement-max-width)` *size space-req* [*Function*]
⇒ `space-requirement-min-width` *space-req* [*Function*]
⇒ `(setf space-requirement-min-width)` *size space-req* [*Function*]
⇒ `space-requirement-height` *space-req* [*Function*]
⇒ `(setf space-requirement-height)` *size space-req* [*Function*]
⇒ `space-requirement-max-height` *space-req* [*Function*]
⇒ `(setf space-requirement-max-height)` *size space-req* [*Function*]
⇒ `space-requirement-min-height` *space-req* [*Function*]
⇒ `(setf space-requirement-min-height)` *size space-req* [*Function*]

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

Part VIII

Appendices

Appendix A

Glossary

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

adaptive toolkit *n.* —Fill this in—

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

affine transformation *n.* A *transformation.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

device transformation *n.* —Fill this in—

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

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

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

display medium *n.* —Fill this in—

display server *n.* —Fill this in—

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

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

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

enabled *adj.* (of a *sheet*) —Fill this in—

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

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

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

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

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

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

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

frame *n.* An *application frame*.

frame manager *n.* An object that controls the realization of the look and feel of an *application frame*.

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

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

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

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

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

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

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

implementor *n.* A programmer who implements CLIM.

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

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

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

input context *n.* 1. —Fill this in—. 2. A Lisp object that represents an *input context*.

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

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

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

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

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

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

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

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

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

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

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

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

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

output recording *n.* The process of remembering the output performed to a *stream*.

output recording stream *n.* A CLIM stream that supports *output recording*.

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

pane *n.* A *sheet* or window that appears as the child of some other window or *frame*. A composite pane can hold other panes; a leaf pane cannot.

parent *n.* The direct ancestor of a *sheet* or an *output record*.

path *n.* A *region* that has dimensionality 1, that is, has length.

patterning *n.* The process of creating a bounded rectangular arrangement of *designs*, like a checkerboard. A *pattern* is a *design* created by this process.

point *n.* 1. A *region* that has dimensionality 0, that is, has only a position. 2. A Lisp object that represents a *point*.

pointer *n.* A physical device used for pointing, such as a mouse.

pointer documentation *n.* —Fill this in—

port *n.* An abstract connection to a *display server* that is responsible for managing host display server resources and for processing input events received from the host display server.

position *n.* 1. A position on a plane, such as CLIM's abstract drawing plane. 2. Two real number values x and y that represent a *position*.

presentation *n.* 1. An association between an object and a *presentation type* with some output on a *output recording stream*. 2. A Lisp object that represents a *presentation*.

presentation tester *n.* A predicate that restricts the applicability of a *presentation translator*.

presentation translator *n.* A mapping from an object of one *presentation type*, an *input context*, and a *gesture* to an object of another presentation type.

presentation type *n.* 1. A description of a class of *presentations*. 2. An extension to CLOS that implements this.

presentation type specifier *n.* A Lisp object used to specify a *presentation type*.

programmer *n.* A person who writes application programs using CLIM.

protocol class *n.* —Fill this in—

rectangle *n.* 1. A four-sided polygon whose sides are parallel to the coordinate axes. 2. A Lisp object that represents a *rectangle*.

redisplay *n.* See *incremental redisplay*.

region *n.* 1. A set of mathematical points in the plane; a mapping from an (x, y) pair into either true or false (meaning member or not a member, respectively, of the region). In CLIM, all regions include their boundaries (that is, they are closed) and have infinite resolution. 2. A Lisp object that represents a *region*.

region set *n.* 1. A "compound" *region*, that is, a region consisting of several other regions related by one of the operations union, intersection, or difference. 2. A Lisp object that represents a *region set*.

rendering *n.* —Fill this in—

replaying *n.* The process of redrawing a set of *output records*.

repainting *n.* —Fill this in—

sensitive *adj.* (of a *presentation*) A *presentation* is *sensitive* if some action will take place when the user clicks on it with the pointer, that is, there is at least one *presentation translator* that is *applicable*. In this case, the presentation will usually be *highlighted*.

server path *n.* —Fill this in—

sheet *n.* 1. —Fill this in—. 2. A Lisp object that represents a *sheet*.

sheet region *n.* —Fill this in—

sheet transformation *n.* —Fill this in—

solid design *n.* A *design* that is either completely opaque or completely transparent. A solid design can be opaque at some points and transparent at others.

stencil *n.* A kind of *pattern* that contains only *opacities*.

stencil opacity *n.* The *opacity* at one point in a *design* that would result from drawing the design onto a fictitious medium whose drawing plane is initially completely transparent black (opacity and all color components are zero), and whose foreground and background are both opaque black. The *stencil opacity* of an *opacity* is simply its value.

stream *n.* A kind of *sheet* that implements the stream protocol (such as maintaining a *text cursor*).

text cursor *n.* —Fill this in—

text style *n.* 1. A description of how textual output should appear, consisting of family, face code, and size. 2. A Lisp object that represents a *text style*.

tiling *n.* The process of repeating a rectangular portion of a *design* throughout the drawing plane. A *tile* is a *design* created by this process.

transformation *n.* 1. A mapping from one coordinate system onto another that preserves straight lines. General transformations include all the sorts of transformations that CLIM uses, namely, translations, scaling, rotations, and reflections. 2. A Lisp object that represents a *transformation*.

translucent design *n.* A *design* that is not *solid*, that is, has at least one point with an opacity that is intermediate between completely opaque and transparent.

true *n.* 1. The boolean value true; not *false*. 2. Any Lisp object that is not `nil`.

unbounded design *n.* A *design* that has at least one point of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.

unbounded region *n.* A *region* that either contains no points or contains points arbitrarily far apart.

uniform design *n.* A *design* that has the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.

unique id *n.* During *incremental redisplay*, the *unique id* is an object used to uniquely identify a piece of output. The output named by the *unique id* will often have a *cache value* associated with it.

user *n.* A person who uses an application program that was written using CLIM.

user transformation *n.* —Fill this in—

view *n.* 1. —Fill this in—. 2. A Lisp object that represents a *view*.

viewport *n.* The portion of the drawing plane of a sheet's medium that is visible on a display device.

volatile *adj.* 1. (of an immutable object) Having components that cannot be modified by the programmer at the protocol level, but which may be modified internally by CLIM. *Volatile* objects reflect internal state of CLIM.

Appendix B

The CLIM-SYS Package

The `clim-sys` package where useful "system-like" functionality lives, including such things as resources and multi-processing primitives. It contains concepts that are not part of Common Lisp, but which are not conceptually the province of CLIM itself.

All of the symbols documented in this appendix must be accessible as external symbols in the `clim-sys` package.

B.1 Resources

⇒ `defresource` *name parameters* `&key` *constructor initializer deinitializer matcher initial-copies* [*Macro*]

Defines a resource named *name*, which must be a symbol. *parameters* is a lambda-list giving names and default values (for optional and keyword parameters) of parameters to an object of this type.

constructor is a form that is responsible for creating an object, and is called when someone tries to allocate an object from the resource and no suitable free objects exist. The constructor form can access the parameters as variables. This argument is required.

initializer is a form that is used to initialize an object gotten from the resource. It can access the parameters as variables, and also has access to a variable called *name*, which is the object to be initialized. The initializer is called both on newly created objects and objects that are being reused.

deinitializer is a form that is used to deinitialize an object when it is about to be returned to the resource. It can access the parameters as variables, and also has access to a variable called *name*, which is the object to be deinitialized. It is called whenever an object is deallocated back to the resource, but is not called by `clear-resource`. Deinitializers are typically used to clear references to other objects.

matcher is a form that ensures that an object in the resource "matches" the specified parameters, which it can access as variables. In addition, the matcher also has access to a variable called *name*, which is the object in the resource being matched against. If no matcher is supplied, the system remembers the values of the parameters (including optional ones that defaulted) that were used to construct the object, and assumes that it matches those particular values for all time. This comparison is done with `equal`. The matcher should return *true* if there is a match, otherwise it should return *false*.

initial-copies is used to specify the number of objects that should be initially put into the resource. It must be an integer or `nil` (the default), meaning that no initial copies should be made. If initial copies are made and there are parameters, all the parameters must be optional; in this case, the initial copies have the default values of the parameters.

⇒ `using-resource` *(variable resource* `&rest` *parameters)* `&body` *body* [*Macro*]

The forms in *body* are evaluated with *variable* bound to an object allocated from the resource named resource, using the given parameters given by *parameters*. The parameters (if any) are evaluated, but *resource* is not.

After the body has been evaluated, `using-resource` returns the object in *variable* back to the resource. If some form in the body sets *variable* to `nil`, the object will not be returned to the resource. Otherwise, the body should not changes the value of *variable*.

⇒ `clear-resource` *resource* [*Function*]

Clears the resource whose name is *resource*, that is, removes all of the resourced object from the resource. *resource* must be a symbol that names a resource.

B.2 Multi-processing

⇒ `make-process` *function* `&key` *name* [*Function*]

Creates a process named *name*. The new process will evaluate the function *function*. On systems that do not support multi-processing, `make-process` will signal an error.

⇒ `destroy-process` *process* [*Function*]

Terminates the process *process*. *process* is an object returned by `make-process`.

⇒ `current-process` [*Function*]

Returns the currently running process, which will be the same kind of object as would be returned by `make-process`.

⇒ `all-processes` [*Function*]

Returns a sequence of all of the processes.

⇒ `process-wait` *reason predicate* [*Function*]

Causes the current process to wait until *predicate* returns *true*. *reason* is a "reason" for waiting, usually a string. On systems that do not support multi-processing, `process-wait` will loop until *predicate* returns *true*.

⇒ `process-wait-with-timeout` *reason timeout predicate* [*Function*]

Causes the current process to wait until either *predicate* returns *true*, or the number of seconds specified by *timeout* has elapsed. *reason* is a "reason" for waiting, usually a string. On systems that do not support multi-processing, `process-wait-with-timeout` will loop until *predicate* returns *true* or the timeout has elapsed.

⇒ `process-yield` [*Function*]

Allows other processes to run. On systems that do not support multi-processing, this does nothing.

⇒ `process-interrupt` *process function* [*Function*]

Interrupts the process *process* and causes it to evaluate the function *function*. On systems that do not support multi-processing, this is equivalent to `funcall`'ing *function*.

⇒ `without-scheduling &body` *body* [*Macro*]

Evaluates *body* in a context that is guaranteed to be free from interruption by other processes. On systems that do not support multi-processing, `without-scheduling` is equivalent to `progn`.

B.3 Locks

⇒ `with-lock-held` *(place* `&optional` *state)* `&body` *body* [*Macro*]

Evaluates *body* with the lock named by *place*. *place* is a reference to a lock created by `make-lock`.

On systems that do not support locking, `with-lock-held` is equivalent to `progn`.

⇒ `make-lock &optional` *name* [*Function*]

Creates a lock whose name is *name*. On systems that do not support locking, this should return a new list of one element, `nil`.

⇒ `with-recursive-lock-held` *(place* `&optional` *state)* `&body` *body* [*Macro*]

Evaluates *body* with the recursive lock named by *place*. *place* is a reference to a recursive lock created by `make-recursive-lock`.

On systems that do not support locking, `with-recursive-lock-held` is equivalent to `progn`.

⇒ `make-recursive-lock &optional` *name* [*Function*]

Creates a recursive lock whose name is *name*. On systems that do not support locking, this should return a new list of one element, `nil`. A recursive lock differs from an ordinary lock in that a process that already holds the recursive lock can call `with-recursive-lock-held` on the same lock without blocking.

Appendix C

Common Lisp Streams

CLIM performs all of its character-based input and output operations on objects called *streams*. Streams are divided into two layers, the *basic stream protocol*, which is character-based and compatible with existing Common Lisp programs, and the *extended stream protocol*, which introduces extended gestures such as pointer gestures and synchronous window-manager communication.

This appendix describes the basic stream-based input and output protocol used by CLIM. The protocol is taken from the `STREAM-DEFINITION-BY-USER` proposal to the X3J13 committee, made by David Gray of TI. This proposal was not accepted by the X3J13 committee as part of the ANSI Common Lisp language definition, but many Lisp implementations do support it. For those implementations that do not support it, it is implemented as part of CLIM.

C.1 Stream Classes

The following classes must be used as superclasses of user-defined stream classes. They are not intended to be directly instantiated; they just provide places to hang default methods.

⇒ `fundamental-stream` [*Class*]

This class is the base class for all CLIM streams. It is a subclass of `stream` and of `standard-object`.

⇒ `streamp` *object* [*Generic Function*]

Returns *true* if *object* is a member of the class `fundamental-stream`. It may return *true* for other objects that are not members of the `fundamental-stream` class, but claim to serve as streams. (It is not sufficient to implement `streamp` as `(typep object 'fundamental-stream)`, because implementations may have additional ways of defining streams.)

⇒ `fundamental-input-stream` [*Class*]

A subclass of `fundamental-stream` that implements input streams.

⇒ `input-stream-p` *object* [*Generic Function*]

Returns *true* when called on any object that is a member of the class `fundamental-input-stream`. It may return *true* for other objects that are not members of the `fundamental-input-stream` class, but claim to serve as input streams.

⇒ `fundamental-output-stream` [*Class*]

A subclass of `fundamental-stream` that implements output streams.

⇒ `output-stream-p` *object* [*Generic Function*]

Returns *true* when called on any object that is a member of the class `fundamental-output-stream`. It may return *true* for other objects that are not members of the `fundamental-output-stream` class, but claim to serve as output streams.

Bidirectional streams can be formed by including both `fundamental-input-stream` and `fundamental-output-stream`.

⇒ `fundamental-character-stream` [*Class*]

A subclass of `fundamental-stream`. It provides a method for `stream-element-type`, which returns `character`.

⇒ `fundamental-binary-stream` [*Class*]

A subclass of `fundamental-stream`. Any instantiable class that includes this needs to define a method for `stream-element-type`.

⇒ `fundamental-character-input-stream` [*Class*]

A subclass of both `fundamental-input-stream` and `fundamental-character-stream`. It provides default methods for several generic functions used for character input.

⇒ `fundamental-character-output-stream` [*Class*]

A subclass of both `fundamental-output-stream` and `fundamental-character-stream`. It provides default methods for several generic functions used for character output.

⇒ `fundamental-binary-input-stream` [*Class*]

A subclass of both `fundamental-input-stream` and `fundamental-binary-stream`.

⇒ `fundamental-binary-output-stream` [*Class*]

A subclass of both `fundamental-output-stream` and `fundamental-binary-stream`.

C.2 Basic Stream Functions

These generic functions must be defined for all stream classes.

⇒ `stream-element-type` *stream* [*Generic Function*]

This existing Common Lisp function is made generic, but otherwise behaves the same. Class `fundamental-character-stream` provides a default method that returns `character`.

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

This function is made generic. A default method is provided by class `fundamental-stream` that returns *true* if `close` has not been called on the stream.

⇒ `close` *stream* `&key` *abort* [*Generic Function*]

The existing Common Lisp function `close` is redefined to be a generic function, but otherwise behaves the same. The default method provided by the class `fundamental-stream` sets a flag used by `open-stream-p`. The value returned by `close` will be as specified by the X3J13 issue `closed-stream-operations`.

⇒ `stream-pathname` *stream* [*Generic Function*]
⇒ `stream-truename` *stream* [*Generic Function*]

These are used to implement `pathname` and `truename`. There is no default method since these are not valid for all streams.

C.3 Character Input

A character input stream can be created by defining a class that includes `fundamental-character-input-stream` and defining methods for the generic functions below.

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

Reads one character from *stream*, and returns either a character object or the symbol `:eof` if the stream is at end-of-file. There is no default method for this generic function, so every subclass of `fundamental-character-input-stream` must define a method.

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

Undoes the last call to `stream-read-char`, as in `unread-char`, and returns `nil`. There is no default method for this, so every subclass of `fundamental-character-input-stream` must define a method.

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

Returns either a character, or `nil` if no input is currently available, or `:eof` if end-of-file is reached.

This is used to implement `read-char-no-hang`. The default method provided by `fundamental-character-input-stream` simply calls `stream-read-char`; this is sufficient for file streams, but interactive streams should define their own method.

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

Returns either a character or `:eof` without removing the character from the stream's input buffer. This is used to implement `peek-char`; this corresponds to peek-type of `nil`. The default method calls `stream-read-char` and `stream-unread-char`.

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

Returns *true* if there is any input pending on *stream*, otherwise it returns *false*. This is used by `listen`. The default method uses `stream-read-char-no-hang` and `stream-unread-char`. Most streams should define their own method since it will usually be trivial and will generally be more efficient than the default method.

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

Returns a string as the first value, and `t` as the second value if the string was terminated by end-of-file instead of the end of a line. This is used by `read-line`. The default method uses repeated calls to `stream-read-char`.

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

Clears any buffered input associated with *stream*, and returns *false*. This is used to implement `clear-input`. The default method does nothing.

C.4 Character Output

A character output stream can be created by defining a class that includes `fundamental-character-output-stream` and defining methods for the generic functions below.

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

Writes *character* to *stream*, and returns *character* as its value. Every subclass of `fundamental-character-output-stream` must have a method defined for this function.

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

This function returns the column number where the next character will be written on *stream*, or `nil` if that is not meaningful. The first column on a line is numbered 0. This function is used in the implementation of `pprint` and the `format ~T` directive. Every character output stream class must define a method for this, although it is permissible for it to always return `nil`.

⇒ `stream-start-line-p` *stream* [*Generic Function*]

Returns *true* if *stream* is positioned at the beginning of a line, otherwise returns *false*. It is permissible to always return *false*. This is used in the implementation of `fresh-line`.

Note that while a value of 0 from `stream-line-column` also indicates the beginning of a line, there are cases where `stream-start-line-p` can be meaningfully implemented when `stream-line-column` cannot. For example, for a window using variable-width characters, the column number isn't very meaningful, but the beginning of the line does have a clear meaning. The default method for `stream-start-line-p` on class `fundamental-character-output-stream` uses `stream-line-column`, so if that is defined to return `nil`, then a method should be provided for either `stream-start-line-p` or `stream-fresh-line`.

⇒ `stream-write-string` *stream string* `&optional` *(start* `0`*) end* [*Generic Function*]

Writes the string *string* to *stream*. If *start* and *end* are supplied, they specify what part of *string* to output. *string* is returned as the value. This is used by `write-string`. The default method provided by `fundamental-character-output-stream` uses repeated calls to `stream-write-char`.

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

Writes an end of line character on *stream*, and returns *false*. This is used by `terpri`. The default method does `stream-write-char` of `#\Newline`.

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

Writes an end of line character on *stream* only if the stream is not at the beginning of the line. This is used by `fresh-line`. The default method uses `stream-start-line-p` and `stream-terpri`.

⇒ `stream-finish-output` *stream* [*Generic Function*]

Ensures that all the output sent to *stream* has reached its destination, and only then return *false*. This is used by `finish-output`. The default method does nothing.

⇒ `stream-force-output` *stream* [*Generic Function*]

Like `stream-finish-output`, except that it may return *false* without waiting for the output to complete. This is used by `force-output`. The default method does nothing.

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

Aborts any outstanding output operation in progress, and returns *false*. This is used by `clear-output`. The default method does nothing.

⇒ `stream-advance-to-column` *stream column* [*Generic Function*]

Writes enough blank space on *stream* so that the next character will be written at the position specified by *column*. Returns *true* if the operation is successful, or `nil` if it is not supported for this stream. This is intended for use by `pprint` and `format ~T`. The default method uses `stream-line-column` and repeated calls to `stream-write-char` with a `#\Space` character; it returns `nil` if `stream-line-column` returns `nil`.

C.5 Binary Streams

Binary streams can be created by defining a class that includes either `fundamental-binary-input-stream` or `fundamental-binary-output-stream` (or both) and defining a method for `stream-element-type` and for one or both of the following generic functions.

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

Returns either an integer, or the symbol `:eof` if *stream* is at end-of-file. This is used by `read-byte`.

⇒ `stream-write-byte` *stream integer* [*Generic Function*]

Writes *integer* to *stream*, and returns *integer* as the result. This is used by `write-byte`.

Appendix D

Encapsulating Streams

An *encapsulating stream* is a special kind of stream that "closes over" another stream, handling some of the usual stream protocol operations itself, and delegating the remaining operations to the "encapsulated" stream. Encapsulating streams may be used by some CLIM implementations in order to facilitate the implementation of features that require the dynamic modification of a stream's state and operations. For example, `accepting-values` dialogs can be implemented by using an encapsulating stream that tailors calls to `accept` and `prompt-for-accept` in such a way that the output is captured and formatted into a dialog that contains prompts and fields that can be clicked on and modified by the user. Input editing can also be implemented using an encapsulating stream that manages the interaction between `read-gesture` and the input editing commands and rescanning. The form `filling-output` can be implemented by having an encapsulating stream that buffers output and inserts line breaks appropriately.

CLIM implementations need not use encapsulating streams at all. If encapsulating streams are used, they must adhere to the following protocols. Encapsulating streams are not part of CLIM's API.

D.1 Encapsulating Stream Classes

⇒ `encapsulating-stream` [*Protocol Class*]

The protocol class that corresponds to an encapsulating stream. If you want to create a new class that behaves like an encapsulating stream, it should be a subclass of `encapsulating-stream`. Subclasses of `encapsulating-stream` must obey the encapsulating stream protocol. Members of this class are mutable.

⇒ `encapsulating-stream-p` *object* [*Function*]

Returns *true* if *object* is an *encapsulating stream*, otherwise returns *false*.

⇒ `:stream` [*Init Arg*]

All encapsulating streams must handle the `:stream` initarg, which is used to specify the stream to be encapsulated.

⇒ `standard-encapsulating-stream` [*Class*]

This class provides a standard implementation of an encapsulating stream.

D.1.1 Encapsulating Stream Protocol

The `standard-encapsulating-stream` class must provide "trampoline" methods for *all* stream protocol operations. These "trampolines" will simply call the same generic function on the encapsulated stream. In particular, all of the generic functions in the following protocols must have trampolines.

- The basic input and output stream protocols, as specified by the Gray stream proposal in Chapter C.
- The sheet protocols, as specified in Chapters 7 and 8.
- The medium protocol, as specified in Chapter 10.
- The text style binding forms, as specified in Chapter 11.
- The drawing functions, as specified in Chapter 12.
- The extended output stream protocol, as specified in Chapter 15.
- The output recording stream protocol, as specified in Chapter 16.
- The incremental redisplay stream protocol, as specified in Chapter 21.
- The extended input stream protocol, as specified in Chapter 22.
- The stream generics for presentation types, as specified in Chapter 23.

The following generic function must also be implemented for all encapsulating stream classes.

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

Returns the stream encapsulated by the *encapsulating stream encapsulating-stream*.

D.1.2 The "Delegation Problem"

The suggested implementation of encapsulating streams has a potential problem that we label the "delegation" or "multiple self" problem. Here is an example of the problem.

Suppose we implement `accepting-values` by using an encapsulating stream class called `accepting-values-stream` that will be used to close over an ordinary extended input and output stream. Let us examine two generic functions, `stream-accept` and `prompt-for-accept`. The `stream-accept` method on an ordinary stream calls `prompt-for-accept`. Now suppose that `accepting-values-stream` specializes `prompt-for-accept`. If we now create a stream of type `accepting-`

`values-stream` (which we will designate A) which encapsulates an ordinary stream S, and then call `stream-accept` on the stream E, it will trampoline to `stream-accept` on the stream S. The desired behavior is for `stream-accept` to call the `prompt-for-accept` method on the stream E, but instead what happens is that the `prompt-for-accept` method on the stream S is called.

In order to side-step this problem without attempting to solve a difficult general problem in object-oriented programming, CLIM implementations may introduce a special variable, `*original-stream*`, which is bound by trampoline functions to the original encapsulating stream. Therefore, the `stream-accept` on the ordinary stream S will call `prompt-for-accept` on the value of `(or *original-stream*` *stream*`)`. This idiom only needs to be used in places where one stream protocol function calls a second stream protocol function that some encapsulating stream specializes.

This "solution" does not solve the more general problem of multiple levels of encapsulation, but the complete stream protocol provided by CLIM should allow implementors to avoid using nested encapsulating streams.

⇒ `*original-stream*` [*Variable*]

This variable is bound by the trampoline methods on encapsulating streams to the encapsulating stream, before the operation is delegated to the underlying, encapsulated stream.

Appendix E

Changes from CLIM 1.0

This appendix lists the incompatible changes from CLIM 1.0 (and CLIM 0.9 for the Silica-related API), and the rationale for those changes. They are listed on a chapter-by-chapter basis.

When the items say that a compatibility stub will be provided, this does not mean that this compatibility needs to be part of CLIM itself. It could be provided by a small compatibility package, or by some sort of conversion utility.

Issue: There are still lots of things from the Silica part and the three panes chapters that need to be included here. — SWM

Geometry

- The use of `region-set-function` has been deprecated in favor of using the three classes `standard-region-union`, `standard-region-intersection`, and `standard-region-difference`, in keeping with the spirit of CLOS. `region-set-function` will be provided as a compatibility function.

Bounding Rectangles

- `with-bounding-rectangle*` used to have optional *max-x* and *max-y* arguments. They are now required.
- The function `bounding-rectangle-set-edges` has been flushed, since bounding rectangles have been made immutable. There is no replacement for it.
- The functions `bounding-rectangle-left`, `bounding-rectangle-top`, `bounding-rectangle-right`, and `bounding-rectangle-bottom` have been replaced by `bounding-rectangle-min-x`, `bounding-rectangle-min-y`, `bounding-rectangle-max-x`, and `bounding-rectangle-max-y`. This is because left, top, right, and bottom are ill-specified. Compatibility functions will be provided.

Affine Transformations

- The function `make-3-point-transformation` has had its argument list changed from *(point-1 point-1-image point-2 point-2-image point-3 point-3-image)* to *(point-1 point-2 point-3 point-1-image point-2-image point-3-image)*. This was done because the original argument list did not group together inputs and output, which was confusing.

- The function `make-3-point-transformation*` has had its argument list changed from *(x1 y1 x1-image y1-image x2 y2 x2-image y2-image x3 y3 x3-image y3-image)* to *(x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image)*. This was done because the original argument list did not group together inputs and output, which was confusing.

- `compose-scaling-transformation`, `compose-translation-transformation`, and `compose-rotation-transformation` have been replaced by the six functions `compose-translation-with-transformation`, `compose-scaling-with-transformation`, `compose-rotation-with-transformation`, `compose-transformation-with-translation`, `compose-transformation-with-scaling`, and `compose-transformation-with-rotation`. This was done because the six functions implement all of the optimized useful cases of composition of transformations, and new names are required for all six. Compatibility functions will be provided for the three CLIM 1.0 functions.

Properties of Sheets

Sheet Protocols

Ports, Grafts, and Mirrored Sheets

Text Styles

- The macros `with-text-style`, `with-text-family`, `with-text-face`, and `with-text-size` have been changed to take the *medium* argument first and the text style (or family, face, or size) argument second. This was done in order to be consistent with all of the other macros that take a *medium* argument as the first argument. Compatibility code will be provided that attempts to detect the old syntax and massages it into the new syntax, although it will probably not be able to detect all cases.

- `add-text-style-mapping` has been replaced by `(setf text-style-mapping)` to be consistent with Common Lisp conventions. A compatibility function will be provided.

Drawing in Color

- `+foreground+` and `+background+` have been renamed to `+foreground-ink+` and `+background-ink+`, for consistency with `+flipping-ink+`. Compatibility constants will be provided.

- `make-color-rgb` and `make-color-ihs` have been renamed to `make-rgb-color` and `make-ihs-color`, by popular demand. Compatibility functions will be provided.

Extended Stream Output

- The function `stream-set-cursor-position*` has been replaced by `(setf stream-cursor-position*)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- The function `stream-vsp` has been replace by `stream-vertical-spacing`. A compatibility function will be provided.
- The macros `with-end-of-line-action` and `with-end-of-page-action` have been changed to take the *stream* argument first and the action argument second. This was done in order to be consistent with all of the other macros that take a *stream* argument as the first argument. Compatibility code will be provided that attempts to detect the old syntax and massages it into the new syntax, although it will probably not be able to detect all cases.

Output Recording

- The three protocol classes `output-record`, `output-record-element`, and `displayed-output-record-element` have been replaced by the two classes `output-record` and `displayed-output-record`. The predicates for the classes have been similarly changed.
- The function `output-record-set-position*` has been replaced by `(setf output-record-position*)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- The functions `output-record-start-position*`, `output-record-set-start-position*`, `output-record-end-position*`, `output-record-set-end-position*` have been replaced by `output-record-start-cursor-position*`, `(setf output-record-start-cursor-position*)`, `output-record-end-cursor-position*`, `(setf output-record-end-cursor-position*)` to better reflect their functionality. Compatibility functions will be provided.
- `replay-1` has been renamed to `replay-output-record`.
- `output-record-elements` and `output-record-element-count` have been renamed to `output-record-children` and `output-record-count`, since we have flushed the term "element". Compatibility functions will be provided.
- `add-output-record-element` and `delete-output-record-element` have been renamed to `add-output-record` and `delete-output-record`, and the argument order has been changed. Compatibility functions will be provided.
- `map-over-output-record-elements-containing-point*` and `map-over-output-record-elements-overlapping-region` have been renamed to `map-over-output-records-containing-point*` and `map-over-output-records-overlapping-region`. Compatibility functions will be provided.

- `linear-output-record` and `coordinate-sorted-set-output-record` have been renamed to `standard-sequence-output-record` and `standard-tree-output-record`.

- `stream-draw-p` and `stream-record-p` and their `setf` functions have been renamed to `stream-drawing-p` and `stream-recording-p` to better reflect their functionality. Compatibility functions will be provided.

- `output-recording-stream-output-record`, `output-recording-stream-current-output-record-stack`, and `output-recording-stream-text-output-record` have been renamed to `stream-output-history`, `stream-current-output-record`, and `stream-text-output-record`. Compatibility functions will be provided.

- `add-output-record` has been renamed to `stream-add-output-record`. Because of the change to `add-output-record-element` above, no compatibility function can be provided.

- `close-current-text-output-record` has been renamed to `stream-close-text-output-record`. A compatibility function will be provided.

- `add-string-output-to-output-record` and `add-character-output-to-output-record` have been renamed to `stream-add-string-output` and `stream-add-character-output`. Compatibility functions will be provided.

- `with-output-recording-options` has had its *draw-p* and *record-p* keyword arguments changed to *draw* and *record* to conform to Common Lisp naming conventions. Compability code will be provided.

Table Formatting

- The `:inter-column-spacing`, `:inter-row-spacing`, and `:multiple-columns-inter-column-spacing` options to `formatting-table` have been renamed to `:x-spacing`, `:y-spacing`, and `:multiple-columns-x-spacing` in order to be consistent with the pane options. Compatibility options will be provided.

- The `:minimum-width` and `:minimum-height` options to `formatting-cell` have been renamed to `:min-width` and :min-height in order to be consistent with the pane options. Compatibility options will be provided.

- The `:inter-column-spacing` and `:inter-row-spacing` options to `formatting-item-list` and `format-items` have been renamed to `:x-spacing` and `:y-spacing` in order to be consistent with the pane options. Compatibility options will be provided.

- The `:no-initial-spacing` option to formatting-item-list and `format-items` has been renamed to `:initial-spacing`, because inverted-sense flags are too hard to keep straight. The default for `:no-initial-spacing` was *true*, therefore the default for `:initial-spacing` is *false*. Compatibility options will be provided.

Graph Formatting

- The function `format-graph-from-root` has been renamed to `format-graph-from-roots`, since it now takes a sequence of root objects, rather than a single root object. The function `format-graph-from-root` will remain as a compatibility function that takes a single root object.

Incremental Redisplay

- `redisplay-1` has been renamed to `redisplay-output-record`.

Extended Stream Input

- The function `stream-set-pointer-position*` has been replaced by `(setf stream-pointer-position*)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- All of the clause arglists for `tracking-pointer` are specified with `&key` , that is, they are named arguments rather than positional ones. This should not cause any problems, except for the one case that the *character* argument to the `:keyboard` clause has been renamed to *gesture*.
- The function `dragging-output-record` has been renamed to `drag-output-record` to be consistent with our naming conventions. A compatibility function will be provided.

Presentation Types

- The argument list for `with-output-as-presentation` has been changed to make *stream*, *object*, and *type* be required arguments instead of keyword arguments. This is because it is always necessary to supply those arguments in order for `with-output-as-presentation` to work. Compability code will be provided to support the old syntax.
- The *activation-characters*, *additional-activation-characters*, *blip-characters*, and *additional-blip-characters* keyword arguments to the `accept` functions have been renamed to *activation-gestures*, *additional-activation-gestures*, *blip-gestures*, and *additional-blip-gestures*. Compability code will be provided to support the old keyword arguments.
- The arglists for presentation translators and their documentation and tester components have been changed to take a single position *object* argument and a list of named (keyword) arguments. Except for translators that omit the *object* argument or have it in other than the initial position of the arglist, this will not pose a problem. This change can be detected.
- The *shift-mask* keyword argument to `test-presentation-translator`, `find-applicable-translators`, `presentation-matches-context-type`, and `find-innermost-applicable-presentation` has been renamed to *modifier-state* in order to be consistent with the device event terminology. Compability code will be provided to support the old keyword.
- The *shifts* keyword argument to `define-gesture-name` has been renamed to *modifiers* in order to be consistent with the device event terminology. Compability code will be provided to support the old keyword.

- dialog-view and +dialog-view+ have been renamed to textual-dialog-view and +textual-dialog-view+ in order to accurately reflect what they are. Likewise, menu-view and +menu-view+ have been renamed to textual-menu-view and +textual-menu-view+. Compatibility classes and constants will be provided.

Input Editing and Completion Facilities

- *activation-characters*, *standard-activation-characters*, with-activation-characters, and activation-character-p have been renamed to *activation-gestures*, *standard-activation-gestures*, with-activation-gestures, and activation-gesture-p. Compatibility functions will remain for with-activation-characters and activation-character-p, but since the variables were not previously documented, no compatibility will be provided for them.
- *blip-characters*, with-blip-characters, and blip-character-p have been renamed to *blip-gestures*, with-blip-gestures, and blip-gesture-p. Compatibility functions will remain for with-blip-characters and blip-character-p, but since *blip-characters* was not previously documented, no compatibility will be provided.
- *abort-characters* has been renamed to *abort-gestures*.
- *completion-characters*, *help-characters*, and *possibilities-characters* have been renamed to *completion-gestures*, *help-gestures*, and *possibilities-gestures*.
- Input editing streams no longer use the interactive stream class. Instead, interactive streams are defined to be any stream that can potentially support input editing, and the class input-editing-stream now refers to input editor streams.
- input-editor-buffer, input-position, insertion-pointer, and rescanning-p have been renamed to stream-input-buffer, stream-scan-pointer, stream-insertion-pointer, and stream-rescanning-p. Compatibility functions will be provided.

Menus

- The :inter-column-spacing and :inter-row-spacing options to menu-choose have been renamed to :x-spacing and :y-spacing in order to be consistent with the pane options. Compatibility options will be provided.

Command Processing

- The variable *unsupplied-argument* has been renamed to *unsupplied-argument-marker* in keeping with its functionality, and to match the new *numeric-argument-marker*. *unsupplied-argument* will be retained, but its use is deprecated.

- The `:inter-column-spacing` and `:inter-row-spacing` options to `display-command-table-menu` have been renamed to `:x-spacing` and `:y-spacing` in order to be consistent with the pane options. Compatibility options will be provided.
- The `:test` argument to the following functions has been removed, since the use of gesture names makes it unnecessary: `add-command-to-command-table`, `add-keystroke-to-command-table`, `remove-keystroke-from-command-table`, `find-keystroke-item`, `lookup-keystroke-item`, and `lookup-keystroke-command-item`. The `:keystroke-test` argument has been removed from `read-command` and `read-command-using-keystrokes` for the same reason.

Application Frames

- The function `frame-top-level-window` has been renamed to `frame-top-level-sheet`. A compatibility function will be provided.
- The function `set-frame-layout` has been replaced by `(setf frame-current-layout)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- `command-enabled-p`, `enable-command`, and `disable-command` have been replaced by `command-enabled` and `(setf command-enabled)`. Compatibility functions will be provided.
- `window-set-viewport-position*` has been replaced by (setf window-viewport-position*). A compatibility function will be provided.

Panes

- The pane options `:hs`, `:hs+`, `:hs-`, `:vs`, `:vs+`, and `:vs-` have been replaced by the options `:width`, `:max-width`, `:min-width`, `:height`, `:max-height`, and `:min-height` to be more perspicuous, and to conform the the same options for the formatted output facilities. Compatibility options will be supplied.
- The `:nchars` and `:nlines` pane options have been flushed in favor of an extended syntax to the `:width` and `:height` options.
- The pane layout options `:halign` and `:valign` have been renamed to `:align-x` and `:align-y` to conform with table formatting. Compatibility options will be supplied.
- The pane layout options `:hspace` and `:vspace` have been renamed to `:x-spacing` and `:y-spacing` to conform with table formatting. Compatibility options will be supplied.

Defining New Pane Types

- The term "space req" has been renamed to "space requirement". All of the functions with `space-req` in their names have been renamed to have `space-requirement` instead.
- `make-space-requirement` no longer takes the `:hs` and `:vs` arguments, *et al*. It now takes `:width` and `:height`, *et al*.