Common Lisp Interface Manager
CLIM II Specification

Scott McKay (SWM@Symbolics.COM)
William York (York@Lucid.COM)
with contributions by
John Aspinall (JGA@Symbolics.COM)
Dennis Doughty (Doughty@ILeaf.COM)
Charles Hornig (Hornig@ODI.COM)
Richard Lamson (RLamson%UMAB.BitNet@MITVMA.MIT.EDU)
David Linden (Linden@CRL.DEC.COM)
David Moon (Moon@Cambridge.Apple.COM)
Ramana Rao (rao@PARC.Xerox.COM)
Chris Richardson (cer@Franz.COM)

May 10, 1994

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, John Irwin, Paul Weineke and JonL White, and Kevin Males. Other contributors include Mike McMahon, Paul Robertson, John Hotchkiss, Dave Lowry, Mike Greenwald, Dave Schatsky, and Glenn Adams. Barry Margolin and Richard Billington, acting for the ALU, contributed useful comments as well.

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 improved significantly 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 programming interface that provides a layered set of portable facilities for constructing user interfaces. These include basic windowing, input, output, and graphics services; stream-oriented input and output extended with facilities such as output recording, presentations, and context sensitive input; high level "formatted output" facilities; application building facilities; command processing; 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 Motif or 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, Motif, OpenLook, the Macintosh, CLOE-386/486, and the Next machine.

The programmer using CLIM is insulated from most of 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 programmer 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. 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, as well as the portable development of toolkit libraries that define and implement a particular look and feel.

In addition, CLIM's layered design allows application programs to exclude facilities that they do not use, or reimplement or extend any part of the substrate. To these ends, CLIM is specified and implemented in a layered, modular fashion based on protocols. 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 windowing module of CLIM defines a uniform interface for creating and managing hierarchies of these objects regardless of their sheet class. This layer also provides event management.

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 macros and functions that enable programs to produce neatly formatted tabular and graphical displays with very little 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 by selecting their displayed representation with the pointer. 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 associating 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 Motif or OpenLook. 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 implement as much of the draft X3J13 Common Lisp

as possible, independent of the conformance of individual vendors. (When all Lisp vendors implement X3J13 Common Lisp, the `clim-lisp` package could be eliminated.) `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 functionality that is not part of Common Lisp, but which is 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 a package that programmers can 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.

2.4.1 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 must 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".

2.5 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 region, design, 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 standard-encapsulating-stream))
  (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.

2.6 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.

2.7 Multiple Value `setf`

Some functions in CLIM that return multiple values have `setf` functions associated with them. For example, `output-record-position` returns the position of an output record as two values that correspond to the x and y coordinates. In order to change the position of an output record, the programmer would like to invoke `(setf output-record-position)`. Normally however, `setf` only takes a single value with which to modify the specified place. CLIM provides a "multiple value" version of `setf` that allows an expression that returns multiple values to be used in updating the specified place. In this specification, this facility will be referred to as `setf*` in the guise of function names such as `(setf* output-record-position)`, even though `setf*` is not actually a defined form.

For example, the modifying function for `output-record-position` might be called in either of the following two ways:

```
(setf (output-record-position record) (values nx ny))

(setf (output-record-position record1) (output-record-position record2))
```

The second form works because `output-record-position` itself returns two values.

Some CLIM implementations may not support `setf*` due to restrictions imposed by the underlying Lisp implementation. In this case, programmers may use special "setter" function instead. In the above example, `output-record-set-position` is the "setter" function.

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 `*standard-input*` 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 continuation &key)
  (declare (dynamic-extent continuation)))

(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) continuation &rest drawing-options)
  (with-drawing-options-merged-into-medium (medium drawing-options)
    (funcall continuation 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

Regions

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. This is a subclass of `design` (see Chapter 13).

If you want to create a new class that behaves like a region, it should be a subclass of `region`. All instantiable 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* [*Protocol Predicate*]

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`. All instantiable subclasses of `path` must obey the path protocol.

Constructing a `path` object with no length (via `make-line*`, for example) may canonicalize 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* [*Protocol Predicate*]

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

Note that constructing a `path` object with no length (such as calling `make-line` with two coincident points), for example) may canonicalize it to `+nowhere+`.

⇒ `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`. All instantiable subclasses of `area` must

obey the area protocol.

Note that constructing an `area` object with no area (such as calling `make-rectangle` with two coincident points), for example) may canonicalize it to `+nowhere+`.

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

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

⇒ `coordinate` [*Type*]

The type that represents a coordinate. This must either be `t`, or a subtype of `real`. CLIM implementations may use a more specific subtype of `real`, such as `single-float`, for reasons of efficiency.

All of the specific region classes and subclasses of `bounding-rectangle` will use this type to store their coordinates. However, the constructor functions for the region classes and for bounding rectangles must accept numbers of any type and coerce them to `coordinate`.

⇒ `coordinate` *n* [*Function*]

Coerces the number *n* to be a coordinate.

⇒ `+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-position-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-position-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* [*Protocol Predicate*]

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 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`.

`map-over-region-set-regions` returns `nil`.

⇒ `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

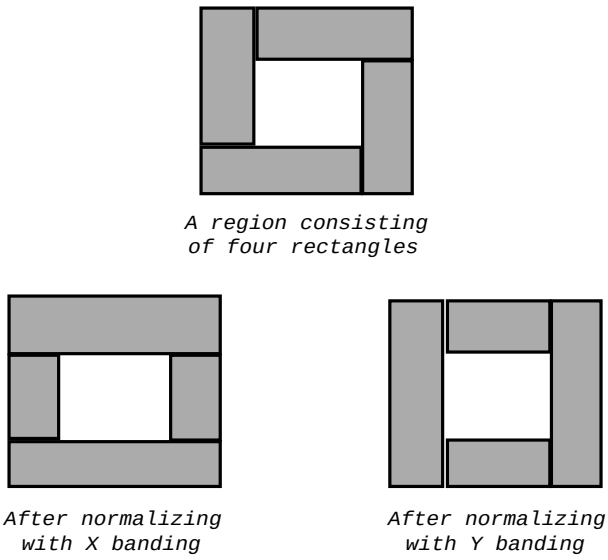

Figure 3.1: Normalization of rectangular region sets.

is a path.

`region-union` will return either a simple region, a region set, 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*]

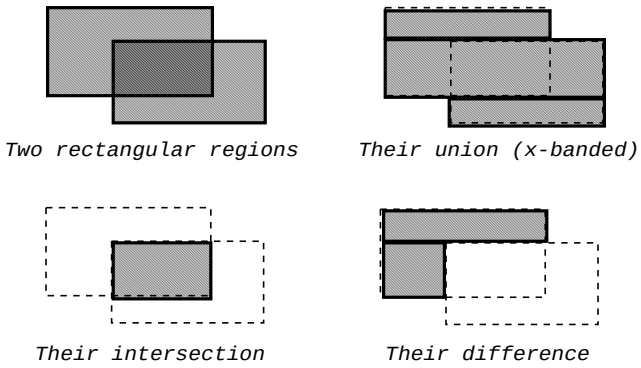

Figure 3.2: Examples of region union, intersection, and difference.

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, a region set, 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.

Major issue: *There is a proposal to remove the* `polygon`, `polyline`, `line`, `ellipse`, *and* `elliptical-arc` *classes, since they are only of limited utility, and CLIM itself doesn't use the classes at all. The advantage of removing these classes is that both the spec and CLIM itself become a little simpler, and there are fewer cases of the region protocol to implement. However, removing these classes results in a geometric model that is no longer closed (in the mathematical sense). This lack of closure makes it difficult to specify the design-based drawing model. Furthermore, these are intuitive objects that are used by a small, but important, class of applications, and some people feel that CLIM should relieve programmers from having to implement these classes for himself or herself.*

The advocates of of removing these classes also propose removing the design-based drawing

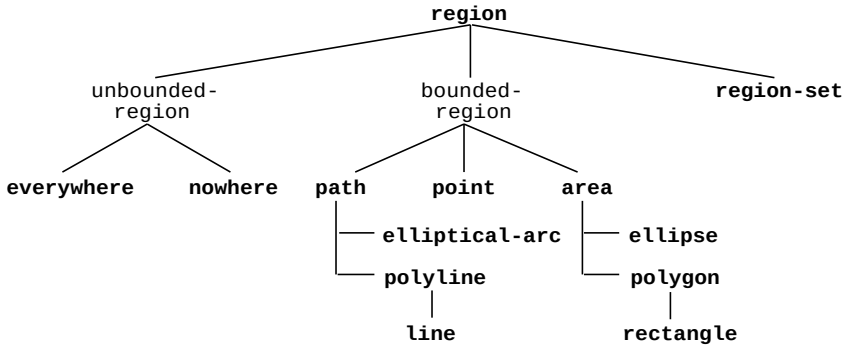

Figure 3.3: The class structure for all regions.

model. In this case, a more consistent proposal is to remove all of the geometric classes, including `point` *and* `rectangle`.

Again, the opposing point of view believes that the power and flexibility of the design-based drawing model does not justify the removal of any of these classes. One counter-proposal is to require CLIM not to use any of the extended region classes internally, and to move the implementation of the extended region classes to a separately loadable module (via `provide` *and* `require`*). — SWM, York*

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`. All instantiable subclasses of `point` must obey the point protocol.

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

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`. All instantiable subclasses of `polyline` must obey the polyline protocol.

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

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`. All instantiable subclasses of `polygon` must obey the polygon protocol.

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

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-coordinates` returns `nil`.

⇒ `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.

`map-over-polygon-segments` returns `nil`.

⇒ `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`. All instantiable subclasses of `line` must obey the line protocol.

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

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`. All instantiable subclasses of `rectangle` must obey the rectangle protocol.

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

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:

	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*]

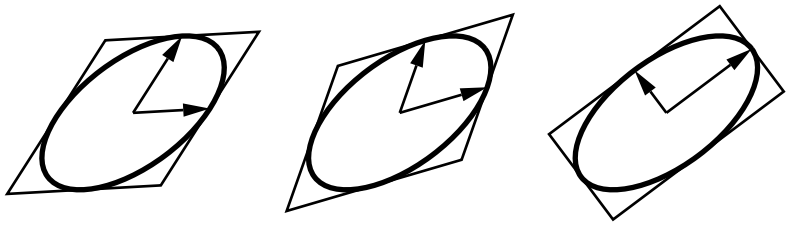

Figure 3.4: Different vectors may specify the same ellipse.

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`. All instantiable subclasses of `ellipse` must obey the ellipse protocol.

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

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`. All instantiable subclasses of `elliptical-arc` must obey the elliptical arc protocol.

⇒ `elliptical-arc-p` *object* [*Protocol Predicate*]

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

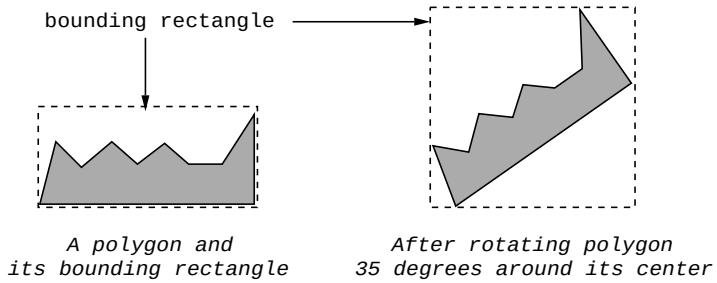

Figure 4.1: The bounding rectangle of an output record.

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.

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`. All instantiable 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* [*Protocol Predicate*]

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`. All instantiable subclasses of `transformation` must obey the transformation protocol.

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

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

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` [*Error Condition*]

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

⇒ `transformation-underspecified` [*Error Condition*]

The error that is signalled when `make-3-point-transformation` is given three collinear image points. This condition will handle the `:points` initarg, which is used to supply the points that are in error.

⇒ `reflection-underspecified` [*Error Condition*]

The error that is signalled when `make-reflection-transformation` is given two coincident points. This condition will handle the `:points` initarg, which is used to supply the points that are in error.

⇒ `singular-transformation` [*Error Condition*]

The error that is signalled when `invert-transformation` is called on a singular transformation, that is, a transformation that has no inverse. This condition will handle the `:transformation` initarg, which is used to supply the transformation that is singular.

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,

and default to 0.

⇒ `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, and default to 0.

⇒ `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:

$$\begin{aligned} x' &= m_{xx}x + m_{xy}y + t_x \\ y' &= m_{yx}x + m_{yy}y + t_y \end{aligned}$$

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

To be supplied.

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

⇒ `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.

Minor issue: *Supply this figure. — SWM*

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.

Note that, if the extended region classes are not implemented, the following functions are not closed, that is, they may return results that are not CLIM regions.

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

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

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

CLIM provides a default method for `untransform-position` 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 *y2* 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 user interfaces is allocating screen regions to particular tasks and recursively subdividing these regions into subregions. The windowing layer of CLIM defines an extensible framework for constructing, using, and managing such *hierarchies of interactive regions*. This framework allows uniform treatment of the following things:

- Window objects like those in X or NeWS.
- Lightweight gadgets typical of toolkit layers, such as Motif or OpenLook.
- 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, CLIM's windowing layer plays the role of a window system. However, CLIM will usually use 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 systems or as display servers.

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

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.

Chapter 7

Properties of Sheets

7.1 Basic Sheet Classes

Note that there are no standard sheet classes in CLIM, and no pre-packaged way to create sheets in general. If a programmer needs to create an instance of some class of sheet, `make-instance` must be used. For most purposes, calling `make-pane` is how application programmers will make sheets.

⇒ `sheet` [*Protocol Class*]

The protocol class that corresponds to a sheet. 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`. All instantiable subclasses of `sheet` must obey the sheet protocol.

All of the subclasses of `sheet` are mutable.

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

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

⇒ `basic-sheet` [*Class*]

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

7.2 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*.

7.2.1 Sheet Relationship Functions

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

⇒ `sheet-parent` *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* `&key` *(errorp* `t`*)* [*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* and *errorp* is *true*, then 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.

⇒ `map-over-sheets` *function sheet* [*Generic Function*]

Applies the function *function* to the sheet *sheet*, and then applies *function* to all of the descendents (the children, the children's children, and so forth) of *sheet*.

Function is a function of one argument, the sheet; it has dynamic extent.

`map-over-sheets` returns `nil`.

7.2.2 Sheet Genealogy Classes

Different "mixin" classes are provided that implement the relationship protocol. None of the four following classes is instantiable.

⇒ `sheet-parent-mixin` [*Class*]

This class is mixed into sheet classes that can act as a parent to other sheets.

⇒ `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. It is a subclass of `sheet-parent-mixin`.

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

This class is mixed into sheet classes that may have zero or more children. It is a subclass of `sheet-parent-mixin`.

7.3 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.

7.3.1 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.

When the sheet's transformation is changed, `note-sheet-transformation-changed` is called on the to notify the sheet of the change.

⇒ `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.

When the sheet's region is changed, `note-sheet-region-region` is called on *sheet* to notify the sheet of the change.

Minor 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*

Minor issue: *I'm not convinced I like this business of requesting a change by modifying an accessor. It might 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. Currently, using* `setf` *on mirrored sheets requests that the server move or resize the sheet; the accessor will continue to return the old value until the notification comes in from the display server that says that the mirror has been moved. — RSL*

⇒ `move-sheet` *sheet x y* [*Generic Function*]

Moves the *sheet sheet* to the new position (x, y). x and y are expressed in the coordinate system of *sheet*'s parent.

`move-sheet` simply modifies *sheet*'s transformation, and could be implemented as follows:

```
(defmethod move-sheet ((sheet basic-sheet) x y)
  (let ((transform (sheet-transformation sheet)))
    (multiple-value-bind (old-x old-y)
        (transform-position transform 0 0)
      (setf (sheet-transformation sheet)
```

```
            (compose-translation-with-transformation
              transform (- x old-x) (- y old-y))))))
```

⇒ `resize-sheet` *sheet width height* [*Generic Function*]

Resizes the *sheet sheet* to have a new width *width* and a new height *height*. *width* and *height* are real numbers.

`resize-sheet` simply modifies *sheet*'s region, and could be implemented as follows:

```
(defmethod resize-sheet ((sheet basic-sheet) width height)
  (setf (sheet-region sheet)
        (make-bounding-rectangle 0 0 width height)))
```

⇒ `move-and-resize-sheet` *sheet x y width height* [*Generic Function*]

Moves the *sheet sheet* to the new position (x, y), and changes its size to the new width *width* and the new height *height*. x and y are expressed in the coordinate system of *sheet*'s parent. *width* and *height* are real numbers.

`move-and-resize-sheet` could be implemented as follows:

```
(defmethod move-and-resize-sheet ((sheet basic-sheet) x y width height)
  (move-sheet sheet x y)
  (resize-sheet sheet width height))
```

⇒ `map-sheet-position-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-position-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-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-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`.

Minor issue: *I now think that* `map-` *in these names is misleading; maybe* `convert-` *is better? — SWM*

⇒ `map-over-sheets-containing-position` *function sheet x y* [*Generic Function*]

Applies the function *function* to all of the children of the sheet *sheet* that contain the position (x, y). x and y are expressed in *sheet*'s coordinate system.

Function is a function of one argument, the sheet; it has dynamic extent.

`map-over-sheets-containing-position` returns `nil`.

⇒ `map-over-sheets-overlapping-region` *function sheet region* [*Generic Function*]

Applies the function *function* to all of the children of the sheet *sheet* that overlap the region *region*. *region* is expressed in *sheet*'s coordinate system.

Function is a function of one argument, the sheet; it has dynamic extent.

`map-over-sheets-overlapping-region` returns `nil`.

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

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

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

Returns the list of enabled direct 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.

⇒ `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.

7.3.2 Sheet Geometry Classes

Each of the following implements the sheet geometry protocol in a different manner, according to the sheet's requirements. None of the four following classes is instantiable.

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

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

⇒ `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. CLIM implementations are allowed to restrict these transformations to just rectilinear ones.

⇒ `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. It is a subclass of `sheet-transformation-mixin`.

⇒ `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. It is a subclass of `sheet-transformation-mixin`.

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.

In addition to handling input event, a sheet is also responsible for providing other input services, such as controlling the pointer's appearance, and polling for current pointer and keyboard state.

Input 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; this process is called *distributing*. 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 *dispatches* 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.

8.1.1 Input Protocol Functions

In the functions below, the *client* argument is typically a sheet, but it may be another object that supports event distribution, dispatching, and handling.

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

Any sheet that can process events will have an event queue from which the events are gotten. `sheet-event-queue` returns the object that acts as the event queue. The exact representation of an event queue is explicitly unspecified.

⇒ `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.

Minor issue: *Should this accessor be called* `keyboard-input-focus`*? It may be that we want to be able to call it on a frame in order to implement some sort of per-frame input focus. — 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.

Minor 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.

⇒ `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. The *client* argument to these functions is typically a sheet.

⇒ `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

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 `pointer-button-click-event`.

None of the five following classes is instantiable.

⇒ `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.

The `standard-sheet-input-mixin` class will also provide a `sheet-event-queue` method, describe above.

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

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

The `immediate-sheet-input-mixin` class will also provide a `sheet-event-queue` method, describe above.

⇒ `sheet-mute-input-mixin` [*Class*]

This is mixed in to any sheet class the does not handle any input events.

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

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

⇒ `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`. *delegate* is the object to which events will be dispatched, and is usually another sheet. 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*]

```
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-boundary-event
               pointer-enter-event
               pointer-exit-event
   window-event
      window-configuration-event
      window-repaint-event
   window-manager-event
      window-manager-delete--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.

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`. All instantiable subclasses of `event` must obey the event protocol.

All of the event classes are immutable. CLIM implementations may choose to keep a resource of the device event classes, but this must be invisible at the API level. That is, any event visible at the level of the API must act as though it is immutable.

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

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

⇒ `:timestamp` [*Initarg*]

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

⇒ `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` [*Initarg*]
⇒ `:modifier-state` [*Initarg*]

The instantiable 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-state` initargs, which are used to specify the sheet and modifier state components for the event.

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

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

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

Returns an integer 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-state`.

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

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

⇒ `pointer-event` [*Class*]
⇒ `:pointer` [*Initarg*]
⇒ `:button` [*Initarg*]
⇒ `:x` [*Initarg*]
⇒ `:y` [*Initarg*]

The instantiable 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.

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

⇒ `pointer-button-event` [*Class*]

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

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

Returns the an integer corresponding to the pointer button that was pressed or released, which will be one of `+pointer-left-button+`, `+pointer-middle-button+`, or `+pointer-right-button+`.

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

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

⇒ `pointer-click-event` [*Class*]
⇒ `pointer-double-click-event` [*Class*]
⇒ `pointer-click-and-hold-event` [*Class*]

The instantiable classes that correspond to a pointer button press, followed immediately by (respectively) a button release, another button press, or pointer motion. These are subclasses of `pointer-button-event`. Ports are not required to generate these events.

⇒ `pointer-motion-event` [*Class*]

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

⇒ `pointer-boundary-event` [*Class*]

The instantiable class that corresponds to a pointer motion event that crosses some sort of sheet boundary. This is a subclass of `pointer-motion-event`.

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

Returns the "kind" of boundary event, which will be one of `:ancestor`, `:virtual`, `:inferior`, `:nonlinear`, `:nonlinear-virtual`, or `nil`. These event kinds correspond to the detail members for X11 enter and exit events.

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

The instantiable classes that correspond to a pointer enter or exit event. These are subclasses of `pointer-boundary-event`.

⇒ `window-event` [*Class*]
⇒ `:region` [*Initarg*]

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

⇒ `window-repaint-event` [*Class*]

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

⇒ `window-manager-event` [*Class*]
⇒ `:sheet` [*Initarg*]

The instantiable class that corresponds to any sort of window manager event. This is a subclass of `event`.

All subclasses of `window-manager-event` must take a `:sheet` initarg, which is used to specify the sheet on which the window manager acted.

⇒ `window-manager-delete-event` [*Class*]

The instantiable class that corresponds to window manager event that causes a host window to be deleted. This is a subclass of `window-manager-event`.

⇒ `timer-event` [*Class*]

The instantiable 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.

These constants are powers of 2 so that they can be combined with `logior` and tested with `logtest`.

⇒ `+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 are powers of 2 that are disjoint from the pointer button constants, so that they can be combined with `logior` and tested with `logtest`.

`event-modifier-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.

8.3 Output Protocol

The output protocol is concerned with the appearance of displayed output on the window associated with a sheet. The sheet output protocol is responsible for providing a means of doing output to a sheet, and for delivering repaint requests to the sheet's client.

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`. All instantiable subclasses of `medium` must obey the medium protocol.

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

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

⇒ `basic-medium` [*Class*]

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

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+`.

Major issue: *What exactly 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 12.7.

Minor issue: *We need to do a little better than this. — SWM*

8.3.3 Output Protocol Classes

The following classes implement the standard output protocols. None of the five following classes is instantiable.

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

This class is mixed in to any sheet that provides the standard output protocol, such as repainting and graphics.

⇒ `sheet-mute-output-mixin` [*Class*]

This class is mixed in to any sheet that provides none of the output protocol.

⇒ `sheet-with-medium-mixin` [*Class*]

This class is used for any sheet that has either a permanent or a temporary medium.

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

This class is mixed in to any sheet that always has a medium associated with it. It is a subclass of `sheet-with-medium-mixin`.

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

This class is mixed in to any sheet that may have a medium associated with it, but does not necessarily have a medium at any given instant. It is a subclass of `sheet-with-medium-mixin`.

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. Only sheets that are subclasses of `sheet-with-medium-mixin` may have a medium associated with them.

⇒ `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 and associated with the sheet for the duration of the body (by calling `engraft-medium`), and then degrafted from the sheet and deallocated when the body has been exited. The values of the last form of the body are returned as the values of `with-sheet-medium`.

This macro will signal a runtime error if sheet is not a subclass of `sheet-with-medium-mixin`.

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 grafted to the sheet, and `with-sheet-medium-bound` will simple evaluate the body. 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`.

This macro will signal a runtime error if sheet is not a subclass of `sheet-with-medium-mixin`.

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`.

This function will signal an error if sheet is not a subclass of `sheet-with-medium-mixin`.

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

Returns the sheet associated with the *medium medium*. If *medium* is not grafted to a sheet, `medium-sheet` returns `nil`.

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

Returns an implementation-dependent object that corresponds to the actual host window that will be drawn on when the *medium medium* is drawn on. If *medium* is not grafted to a sheet or the medium's sheet is not currently mirrored on a display server, `medium-drawable` returns `nil`.

Programmers can use this function to get a host window system object that can be manipulated using the functions of the host window system. This might be done in order to explicitly trade of performance against portability.

⇒ `port` *(medium* `basic-medium`*)* [*Method*]

If *medium* is both grafted to a sheet and the sheet is currently mirrored on a display server, this returns the port with which *medium* is associated. Otherwise it returns `nil`.

Grafting and Degrafting of Mediums

The following generic functions are the protocol-level functions responsible for the allocating, deallocating, grafting, and degrafting of mediums. They are not intended for general use by programmers.

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

Allocates a medium from the *port port*'s medium resource, or calls `make-medium` on the port to create a new medium if the resource is empty or the port does not maintain a resource of mediums. The resulting medium will have its default characteristics determined by *sheet*.

⇒ `deallocate-medium` *port medium* [*Generic Function*]

Returns the *medium medium* to *port*'s medium resource.

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

Creates a new medium for the *port port*. The new medium will have its default characteristics determined by *sheet*.

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

Grafts the *medium medium* to the *sheet sheet* on the *port port*.

The default method on `basic-medium` will set `medium-sheet` on *medium* to point to the sheet, and will set up the medium state (foreground ink, background ink, and so forth) from the defaults gotten from sheet. Each implementation may specialize this generic function in order to set up such things as per-medium ink caches, and so forth.

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

Degrafts the *medium medium* from the *sheet sheet* on the *port port*.

The default method on `basic-medium` will set `medium-sheet` back to `nil`. Each implementation may specialize this generic function in order to clear any caches it has set up, and so forth.

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

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

Requests that the repaint event *repaint-event* 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 sheet using the repaint region gotten from *repaint-event*.

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

Implements repainting for a given sheet class. *sheet* is the sheet to repaint and *region* is the region to repaint.

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

Recursively causes repainting of the *sheet sheet* and any of its descendants that overlap the *region region*.

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

8.4.2 Repaint Protocol Classes

The following classes implement the standard repaint protocols. None of the three following classes is instantiable.

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

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

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

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

⇒ `sheet-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

Minor issue: *More 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 Sheet Geometry Notifications

Minor issue: *More 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.

⇒ `port` [*Protocol Class*]

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

All of the subclasses of `port` are mutable.

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

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

⇒ `basic-port` [*Class*]

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

⇒ `find-port &rest` *initargs* `&key` *(server-path* `*default-server-path*` *)* `&allow-other-keys` [*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 initargs in *initargs* will be passed to the function that constructed the new port.

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

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

The value of `*default-server-path*` is a cons of a port type followed by a list of initargs.

The following are the recommendations for port types and their initargs. This list is not intended to be comprehensive, nor is it required that a CLIM implementation support any of these port types.

⇒ `:clx &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.

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

Given this server path, `find-port` finds a port for a Motif 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.

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

Given this server path, `find-port` finds a port for an OpenLook 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.

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

Given this server path, `find-port` finds a port for local Genera platform on the screen object *screen*. *screen* defaults to *tv:main-screen*, but could also be an object return from `color:find-color-screen`.

⇒ `port` *object* [*Generic Function*]

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

⇒ `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 `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.

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

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

`map-over-ports` returns `nil`.

⇒ `port-server-path` *port* [*Generic Function*]

Returns the server path associated with the *port port*.

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

Returns an implementation-dependent string that is the name of the port. For example, a `:clx` port might have a name of `"summer:0.0"`.

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

Returns the type of the port, that is, the first element of the server path spec.

⇒ `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.

⇒ `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.

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` *(server-path* `*default-server-path*`*) (port* `(find-port :server-path 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.

Minor issue: *I don't know how much of this is obsolete. — RSL*

⇒ `graft` *object* [*Generic Function*]

Returns the graft currently associated with *object*. `graft` is defined for all sheet classes (including streams that support the CLIM graphics protocol), mediums, and application frames. For degrafted sheets or other objects that aren't currently associated with a particular graft, `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.

`map-over-grafts` returns `nil`.

⇒ `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 `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`.

Minor 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.

⇒ `mirrored-sheet-mixin` [*Class*]

This class is mixed in to sheet classes that can be directly mirrored.

9.4.1 Mirror Functions

Minor issue: *What kind of an object is a mirror? Is it the Lisp object that is the handle to the actual toolkit window or gadget? — SWM*

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

Returns the mirror of the *sheet sheet*. If the sheet is not a subclass of `mirrored-sheet-mixin`, this will return `nil`. If the sheet is a subclass of `mirrored-sheet-mixin` and does not currently have a mirror, `sheet-mirror` will return `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; the type of this object is implementation dependent.

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

Destroys the mirror for the *sheet mirrored-sheet* on the *port port*.

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

Raises the *sheet sheet*'s mirror to the top of all of the host windows on the *port port*. *sheet* need not be a directly mirrored sheet.

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

Buries the *sheet sheet*'s mirror at the bottom of all of the host windows on the *port port*. *sheet* need not be a directly mirrored sheet.

⇒ `port` *(sheet* `basic-sheet`*)* [*Method*]

If *sheet* is currently mirrored on a display server, this returns the port with which *sheet* is associated. Otherwise it returns `nil`.

9.4.2 Internal Interfaces for Native Coordinates

Minor issue: *Do these functions work on any sheet, or only on sheets that have a mirror, or only on sheets that have a direct mirror? Also, define what a "native coordinate" are. Also, do* `sheet-device-transformation` *and* `sheet-device-region` *really account for the user's transformation and clipping region? — SWM*

⇒ `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.

⇒ `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`.

The clipping region is expressed in user coordinates.

⇒ `(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`.

The default value for `medium-default-text-style` for any medium is `*default-text-style*`.

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 *continuation* argument to `invoke-with-drawing-options`. The exact behavior of this macro is described under `invoke-with-drawing-options`.

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

Binds the state of the *medium medium* to correspond to the supplied drawing options, and then calls the function *continuation* with the new drawing options in effect. *continuation* 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.