The `:ink` *ink* drawing option temporarily changes the value of `(medium-ink` *medium*`)` to *ink*, replacing the previous ink; the new and old inks are not combined in any way.

⇒ `:transformation` [*Option*]

This transforms the coordinates used as arguments to drawing functions to the coordinate system of the drawing plane. The default value is `+identity-transformation+`. See Chapter 5 for a complete description of transformations.

The `:transformation` *xform* drawing option temporarily changes the value of`(medium-transformation` *medium*`)` to `(compose-transformations (medium-transformation` *medium*`)` *xform*`)`.

⇒ `:clipping-region` [*Option*]

The drawing functions do not affect the drawing plane outside this region. The clipping region must be an `area`. Furthermore, some implementations might signal an error if the clipping region is not a rectangle or a region set composed entirely of rectangles. Rendering is clipped both by this clipping region and by other clipping regions associated with the mapping from the target drawing plane to the viewport that displays a portion of the drawing plane. The default is `+everywhere+`, or in other words, no clipping occurs in the drawing plane, only in the viewport.

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

Minor issue: *A better explanation is needed. It does the right thing, but it's hard to tell that from this description. That is, the clipping region is expressed in user coordinates. — DCPL*

⇒ `:line-style` [*Option*]

The line and arc drawing functions render according to this line style. The line style suboptions and default are defined in Section 10.3.

The `:line-style` *ls* drawing option temporarily changes the value of `(medium-line-style` *medium*`)` to *ls*, replacing the previous line style; the new and old line styles are not combined in any way.

If line style suboptions are supplied, they temporarily change the value of `(medium-line-style` *medium*`)` to a line style constructed from the specified suboptions. Components not specified by suboptions are defaulted from the `:line-style` drawing option, if it is supplied, or else from the previous value of `(medium-line-style` *medium*`)`. That is, if both the `:line-style` option and line style suboptions are supplied, the suboptions take precedence over the components of the `:line-style` option.

⇒ `:text-style` [*Option*]

The text drawing functions, including ordinary stream output, render text as directed by this text style merged against the default text style. The default value has all null components. See

Chapter 11 for a complete description of text styles, including the text style suboptions.

The `:text-style` *ts* drawing option temporarily changes the value of `(medium-text-style` *medium*`)` to `(merge-text-styles` *ts* `(medium-text-style` *medium*`))`.

If text style suboptions are supplied, they temporarily change the value of `(medium-text-style` *medium*`)` to a text style constructed from the specified suboptions, merged with the `:text-style` drawing option if it is specified, and then merged with the previous value of `(medium-text-style` *medium*`)`. That is, if both the `:text-style` option and text style suboptions are supplied, the suboptions take precedence over the components of the `:text-style` option.

10.2.1 Transformation "Convenience" Forms

The following three functions are no different than using `with-drawing-options` with the `:transformation` keyword argument supplied. However, they are sufficiently useful that they are provided as a convenience to programmers.

In order to preserve referential transparency, these three forms apply the translation, rotation, or scaling transformation first, then the rest of the transformation from `(medium-transformation` *medium*`)`. That is, the following two forms would return the same transformation (assuming that the medium's transformation in the second example is the identity transformation):

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

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

⇒ `with-translation` *(medium dx dy)* `&body` *body* [*Macro*]

Establishes a translation on the *medium* designated by *medium* that translates by *dx* in the *x* direction and *dy* in the *y* direction, and then executes *body* with that transformation in effect.

dx and *dy* are as for `make-translation-transformation`.

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

⇒ `with-scaling` *(medium sx* `&optional` *sy origin)* `&body` *body* [*Macro*]

Establishes a scaling transformation on the *medium* designated by *medium* that scales by *sx* in the *x* direction and *sy* in the *y* direction, and then executes *body* with that transformation in effect. If *sy* is not supplied, it defaults to *sx*. If *origin* is supplied, the scaling is about that point; if it is not supplied, it defaults to $(0, 0)$.

sx and *sy* are as for `make-scaling-transformation`.

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

⇒ `with-rotation` *(medium angle* `&optional` *origin)* `&body` *body* [*Macro*]

Establishes a rotation on the *medium* designated by *medium* that rotates by *angle*, and then executes *body* with that transformation in effect. If *origin* is supplied, the rotation is about that point; if it is not supplied, it defaults to $(0, 0)$.

angle and *origin* are as for `make-rotation-transformation`.

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

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

Establishes the identity transformation on the *medium* designated by *medium*.

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

10.2.2 Establishing Local Coordinate Systems

⇒ `with-local-coordinates` *(medium* `&optional` *x y)* `&body` *body* [*Macro*]

Binds the dynamic environment to establish a local coordinate system on the *medium* designated by *medium* with the origin of the new coordinate system at the position (x, y). The "directionality" of the coordinate system is otherwise unchanged. x and y are real numbers, and both default to 0.

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

⇒ `with-first-quadrant-coordinates` *(medium* `&optional` *x y)* `&body` *body* [*Macro*]

Binds the dynamic environment to establish a local coordinate system on the *medium* designated by *medium* with the positive x axis extending to the right and the positive y axis extending upward, with the origin of the new coordinate system at the position (x, y). x and y are real numbers, and both default to 0.

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

10.3 Line Styles

A line or other path is a one-dimensional object. However in order to be visible, the rendering of a line must occupy some non-zero area on the display hardware. A *line style* represents the advice of CLIM to the rendering substrate on how to perform the rendering.

⇒ `line-style` [*Protocol Class*]

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

⇒ `line-style-p` *object* [*Protocol Predicate*]

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

⇒ `standard-line-style` [*Class*]

An instantiable class that implements line styles. A subclass of `line-style`. This is the class that `make-line-style` instantiates. Members of this class are immutable.

⇒ `make-line-style &key` *unit thickness joint-shape cap-shape dashes* [*Function*]

Returns an object of class `standard-line-style` with the supplied characteristics. The arguments and their default values are described in Section 10.3.1.

10.3.1 Line Style Protocol and Line Style Suboptions

Each of these suboptions has a corresponding reader that can be used to extract a particular component from a line style. The generic functions decribed below comprise the line style protocol; all subclasses of `line-style` must implement methods for these generic functions.

⇒ `:line-unit` [*Option*]
⇒ `line-style-unit` *line-style* [*Generic Function*]

Gives the unit used for measuring line thickness and dash pattern length for the line style. Possible values are `:normal`, `:point`, or `:coordinate`. The meaning of these options is:

- `:normal`—thicknesses and lengths are given in a relative measure in terms of the usual or "normal" line thickness. The normal line thickness is the thickness of the "comfortably visible thin line", [1] which is a property of the underlying rendering substrate. This is the default.

[1] In some window systems, the phrase "thinnest visible line" is used. This is not appropriate for CLIM, which intends to be device independent. (For instance, the thinnest visible line on a 400 d.p.i. laser printer is a function of the user's viewing distance from the paper.) Another attribute of a "normal" line is that its thickness should approximately match the stroke thickness of "normal" text, where again the exact measurements are the province of the rendering engine, not of CLIM.

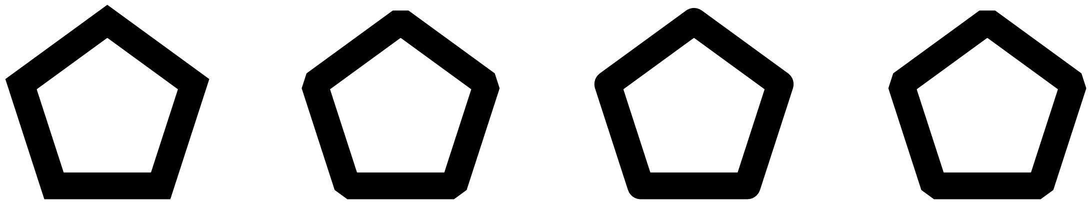


Figure 10.1: Line joint shapes.

- `:point`—thicknesses and lengths are given in an absolute measure in terms of printer's points (approximately 1/72 of an inch). [2]
- `:coordinate`—this means that the same units should be used for line thickness as are used for coordinates. In this case, the line thickness is scaled by the medium's current transformation, whereas `:normal` and `:point` do not scale the line thickness.

⇒ `:line-thickness` [*Option*]
⇒ `line-style-thickness` *line-style* [*Generic Function*]

The thickness, in the units indicated by `line-style-unit`, of the lines or arcs drawn by a drawing function. The thickness must be a real number. The default is 1, which combined with the default unit of `:normal`, means that the default line drawn is the "comfortably visible thin line".

⇒ `:line-joint-shape` [*Option*]
⇒ `line-style-joint-shape` *line-style* [*Generic Function*]

Specifies the shape of joints between segments of unfilled figures. The possible shapes are `:miter`, `:bevel`, `:round`, and `:none`; the default is `:miter`. Note that the joint shape is implemented by the host window system, so not all platforms will necessarily fully support it.

⇒ `:line-cap-shape` [*Option*]
⇒ `line-style-cap-shape` *line-style* [*Generic Function*]

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

⇒ `:line-dashes` [*Option*]
⇒ `line-style-dashes` *line-style* [*Generic Function*]

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

[2] This measure was chosen so that CLIM implementors who interface CLIM to an underlying rendering engine (the window system) may legitimately choose to make it render as 1 pixel on current (1990) display devices.

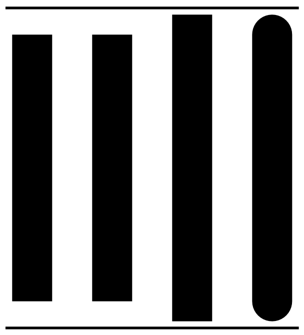


Figure 10.2: Line cap shapes.

- `nil`—lines are drawn solid, with no dashing. This is the default.
- `t`—lines are drawn dashed, with a dash pattern that is unspecified and may vary with the rendering engine. This allows the underlying display substrate to provide a default dashed line for the programmer whose only requirement is to draw a line that is visually distinguishable from the default solid line.
- A sequence—specifies a sequence, usually a vector, controlling the dash pattern of a drawing function. It is an error if the sequence does not contain an even number of elements. The elements of the sequence are lengths (as real numbers) of individual components of the dashed line or arc. The odd elements specify the length of inked components, the even elements specify the gaps. All lengths are expressed in the units described by `line-style-unit`.

(See also `make-contrasting-dash-patterns`.)

10.3.2 Contrasting Dash Patterns

⇒ `make-contrasting-dash-patterns` *n* `&optional` *k* [*Function*]

If *k* is not supplied, this returns a vector of *n* dash patterns with recognizably different appearance. Elements of the vector are guaranteed to be acceptable values for `:dashes`, and do not include `nil`, but their class is not otherwise specified. The vector is a fresh object that may be modified.

If *k* is supplied, it must be an integer between 0 and $n - 1$ (inclusive), in which case `make-contrasting-dash-patterns` returns the *k*'th dash-pattern rather than returning a vector of dash-patterns.

If the implementation does not have *n* different contrasting dash patterns, `make-contrasting-dash-patterns` signals an error. This will not happen unless *n* is greater than eight.

⇒ `contrasting-dash-pattern-limit` *port* [*Generic Function*]

Returns the number of contrasting dash patterns that can be rendered on any medium on the *port port*. Implementations are encouraged to make this as large as possible, but it must be at least 8. All classes that obey the port protocol must implement a method for this generic function.

Chapter 11

Text Styles

When specifying a particular "appearance" for rendered characters, there is a tension between portability and access to specific font for a display device. CLIM provides a portable mechanism for describing the desired *text style* in abstract terms. Each CLIM "port" defines a mapping between these abstract style specifications and particular device-specific fonts. In this way, an application programmer can specify the desired text style in abstract terms secure in the knowledge that an appropriate device font will be selected at run time by CLIM. However, some programmers may require direct access to particular device fonts. The text style mechanism supports specifying device fonts by name, allowing the programmer to sacrifice portability for control.

11.1 Text Styles

Text style objects have components for family, face, and size. Not all of these attributes need be supplied for a given text style object. Text styles can be merged in much the same way as pathnames are merged; unspecified components in the style object (that is, components that have `nil` in them) may be filled in by the components of a "default" style object. A text style object is called *fully specified* if none of its components is `nil`, and the size component is not a relative size (that is, is neither `:smaller` nor `:larger`).

⇒ `text-style` [*Protocol Class*]

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

⇒ `text-style-p` *object* [*Protocol Predicate*]

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

⇒ `standard-text-style` [*Class*]

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

The interface to text styles is as follows:

⇒ `make-text-style` *family face size* [*Function*]

Returns an object of class `standard-text-style` with a family of *family*, a face of *face*, and a size of *size*.

family is one of `:fix`, `:serif`, `:sans-serif`, or `nil`.

face is one of `:roman`, `:bold`, `:italic`, `(:bold :italic)`, or `nil`.

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

Implementations are permitted to extend legal values for *family*, *face*, and *size*.

Minor issue: *Need to describe what family, face, size mean in terms of visual appearance. This should also be reconciled with the ISO description of the attributes of a "text style", including such things as underlining, subscripts, superscripts, etc. — York, SWM*

⇒ `*default-text-style*` [*Constant*]

The default text style used on a CLIM medium if no text style it explicitly specified for the medium when it it created. This must be a fully merged text style.

⇒ `*undefined-text-style*` [*Constant*]

The text style that is used as a fallback if no mapping exists for some other text style when some text is about to be rendered on a display device (via `write-string` and `draw-string*`, for example). This text style be fully merged, and it must have a mapping for all display devices.

11.1.1 Text Style Protocol and Text Style Suboptions

The following generic functions comprise the text style protocol. All subclasses of `text-style` must implement methods for each of these generic functions.

Each of the suboptions described below has a corresponding reader accessor that can be used to extract a particular component from a text style.

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

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

⇒ `:text-family` [*Option*]
⇒ `text-style-family` *text-style* [*Generic Function*]

Specifies the family of the *text style text-style*.

⇒ `:text-face` [*Option*]
⇒ `text-style-face` *text-style* [*Generic Function*]

Specifies the face of the *text style text-style*.

⇒ `:text-size` [*Option*]
⇒ `text-style-size` *text-style* [*Generic Function*]

Specifies the size of the *text style text-style*.

⇒ `parse-text-style` *style-spec* [*Function*]

Returns a text style object. *style-spec* may be a `text-style` object or a device font, in which case it is returned as is, or it may be a list of the family, face, and size (that is, a "style spec"), in which case it is "parsed" and a `text-style` object is returned. This function is for efficiency, since a number of common functions that take a style object as an argument can also take a style spec, in particular `draw-text`.

⇒ `merge-text-styles` *style1 style2* [*Generic Function*]

Merges the *text styles style1* with *style2*, that is, returns a new text style that is the same as *style1*, except that unspecified components in *style1* are filled in from *style2*. For convenience, the two arguments may be also be style specs.

When merging the sizes of two text styles, if the size from *style1* is a relative size, the resulting size is either the next smaller or next larger size than is specified by *style2*. The ordering of sizes, from smallest to largest, is `:tiny`, `:very-small`, `:small`, `:normal`, `:large`, `:very-large`, and `:huge`.

Minor issue: *Need to describe face-merging properly. For example, merging a bold face with an italic one can result in a bold-italic face. — SWM*

⇒ `text-style-ascent` *text-style medium* [*Generic Function*]
⇒ `text-style-descent` *text-style medium* [*Generic Function*]
⇒ `text-style-height` *text-style medium* [*Generic Function*]
⇒ `text-style-width` *text-style medium* [*Generic Function*]

Returns the ascent, descent, height, and width (respectively) of the font corresponding to the *text style text-style* as it would be rendered on the *medium medium*. *text-style* must be a fully specified text style.

The ascent of a font is the distance between the top of the tallest character in that font and the font's baseline. The descent of a font is the distance between the baseline and the bottom of the lowest descending character (usually "g", "p", "q", or "y"). The height of a font is the sum of the ascent and the descent of the font. The width of a font is the width of some representative character in the font.

The methods for these generic functions will typically specialize both the *text-style* and *medium* arguments. Implementations should also provide "trampoline" for these generic functions on output sheets; the trampolines will simply call the method for the medium.

⇒ `text-style-fixed-width-p` *text-style medium* [*Generic Function*]

Returns *true* if the *text styles text-style* will map to a fixed-width font on the *medium medium*, otherwise returns *false*. *text-style* must be a fully specified text style.

The methods for this generic function will typically specialize both the *text-style* and *medium* arguments. Implementations should also provide a "trampoline" for this generic function for output sheets; the trampoline will simply call the method for the medium.

Minor issue: *Discuss baselines? Kerning? — SWM*

⇒ `text-size` *medium string* `&key` *text-style (start* `0`*) end* [*Generic Function*]

Computes the "cursor motion" in device units that would take place if *string* (which may be either a string or a character) were output to the *medium medium* starting at the position $(0,0)$. Five values are returned: the total width of the string in device units, the total height of the string in device units, the final x cursor position (which is the same as the width if there are no `#\Newline` characters in the string), the final y cursor position (which is 0 if the string has no `#\Newline` characters in it, and is incremented by the line height of *medium* for each `#\Newline` character in the string), and the string's baseline.

text-style specifies what text style is to be used when doing the output, and defaults to `medium-merged-text-style` of the medium. *text-style* must be a fully specified text style. *start* and *end* may be used to specify a substring of *string*.

If a programmer needs to account for kerning or the ascent or descent of the text style, he should measure the size of the bounding rectangle of the text rendered on *medium*.

All mediums and output sheets must implement a method for this generic function.

11.2 Text Style Binding Forms

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

Binds the current text style of the *medium* designated by *medium* to correspond to the new text style. *text-style* may either a text style object or a style spec (that is, a list of the a family, a face code, and a size). *body* is executed with the new text style in effect.

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

`with-text-style` must be implemented by expanding into a call to `invoke-with-text-style`, supplying a function that executes *body* as the *continuation* argument to `invoke-with-text-style`.

⇒ `invoke-with-text-style` *medium continuation text-style* [*Generic Function*]

Binds the current text style of the *medium medium* to correspond to the new text style, and calls the function *continuation* with the new text style in effect. *text-style* may either a text style object or a style spec (that is, a list of the a family, a face code, and a size). *continuation* is a function of one argument, the medium; it has dynamic extent.

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

⇒ `with-text-family` *(medium family)* `&body` *body* [*Macro*]
⇒ `with-text-face` *(medium face)* `&body` *body* [*Macro*]
⇒ `with-text-size` *(medium size)* `&body` *body* [*Macro*]

Binds the current text style of the *medium* designated by *medium* to correspond to a new text style consisting of the current text style with the new family, face, or size (respectively) merged in. *face*, *family*, and *size* are as for `make-text-style`. *body* is executed with the new text style in effect.

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

These macros are "convenience" forms of `with-text-style` that must expand into calls to `invoke-with-text-style`.

11.3 Controlling Text Style Mappings

Text styles are mapped to fonts using the `text-style-mapping` function, which takes a port, a character set, and a text style and returns a font object. All ports must implement methods for the following generic functions, for all classes of text style.

The objects used to represent a font mapping are unspecified and are likely to vary from port to port. For instance, a mapping might be some sort of font object on one type of port, or might simply be the name of a font on another.

Minor issue: *We still need to describe what a device font is. Ditto, character sets. — SWM*

Part of initializing a port is to define the mappings between text styles and font names for the port's host window system.

⇒ `text-style-mapping` *port text-style* `&optional` *character-set* [*Generic Function*]

Returns the font mapping that will be used when rendering characters in the character set *character-set* in the *text style text-style* on any medium on the *port port*. If there is no mapping associated with *character-set* and *text-style* on *port*, then some other object will be returned that corresponds to the "unmapped" text style.

A port may implement either an "exact" or a "loose" text style mapping scheme. If the port is using exact text style mappings, `text-style-mapping` will return a font whose size exactly matches the size specified in *text-style*. Otherwise if the port is using loose text style mappings, `text-style-mapping` will return the font whose size is closest to the size specified in *text-style*.

character-set defaults to the standard character set.

⇒ `(setf text-style-mapping)` *mapping port text-style* `&optional` *character-set* [*Generic Function*]

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

character-set defaults to the standard character set.

⇒ `make-device-font-text-style` *display-device device-font-name* [*Function*]

Returns a text style object that will be mapped directly to the specified device font when text is output to a to the display device with this style. Device font styles do not merge with any other kind of style.

Chapter 12

Graphics

12.1 Overview of Graphics

The CLIM graphic drawing model is an idealized model of graphical pictures. The model provides the language that application programs use to describe the intended visual appearance of textual and graphical output. Usually not all of the contents of the screen are described using the graphic drawing model. For example, menus and scroll bars might be described in higher-level terms.

An important aspect of the CLIM graphic drawing model is its extreme device independence. The model describes ideal graphical images and ignores limitations of actual graphics devices. One consequence of this is that the actual visual appearance of the screen can only be an approximation of the appearance specified by the model. Another consequence of this is that the model is highly portable.

CLIM separates output into two layers, a text/graphics layer in which one specifies the desired visual appearance independent of device resolution and characteristics, and a rendering layer in which some approximation of the desired visual appearance is created on the device. Of course application programs can inquire about the device resolution and characteristics if they wish and modify their desired visual appearance on that basis. (There is also a third layer above these two layers, the adaptive toolkit layer where one specifies the desired functionality rather than the desired visual appearance.)

Major issue: *There are still no functions to ask about device resolution and characteristics. What characteristics do we need to be able to get to besides the obvious ones of resolution and "color depth". Also, do we really need to refer to the adaptive toolkit layer here? — SWM*

CLIM's drawing functions provide convenient ways to draw several commonly-used shapes.

The interaction between graphics and output recording will be described in Chapter 16.

12.2 Definitions

Drawing plane. A drawing plane is an infinite two-dimensional plane on which graphical output occurs. The drawing plane contains an arrangement of colors and opacities that is modified by each graphical output operation. It is not possible to read back the contents of a drawing plane, except by examining the output-history. Normally each window has its own drawing plane.

Coordinates. Coordinates are a pair of real numbers in implementation-defined units that identify a point in the drawing plane.

Sheets and Mediums. In this chapter, we use a medium as a destination for output. The medium has a drawing plane, two designs called the medium's foreground and background, a transformation, a clipping region, a line style, and a text style. There are per-medium, dynamically scoped, default drawing options. Different medium classes are provided to allow programmers to draw on different sorts of devices, such as displays, printers, and virtual devices such as bitmaps.

Many sheets can be used for doing output, so the drawing functions can also take a sheet as the output argument. In this case, drawing function "trampolines" to the sheet's medium. So, while the functions defined here are specified to be called on sheets, they can also be called on sheets.

A stream is a special kind of sheet that implements the stream protocol; streams include additional state such as the current text cursor (which is some point in the drawing plane).

By default, the "fundamental" coordinate system of a CLIM stream (not a general sheet or medium, whose fundamental coordinate system is not defined) is a left handed system with x increasing to the right, and y increasing downward. $(0, 0)$ is at the upper left corner.

12.3 Drawing is Approximate

Note that although the drawing plane contains an infinite number of mathematical points, and drawing can be described as an infinite number of color and opacity computations, the drawing plane cannot be viewed directly and has no material existence. It is only an abstraction. What can be viewed directly is the result of rendering portions of the drawing plane onto a medium. No infinite computations or objects of infinite size are required to implement CLIM, because the results of rendering have finite size and finite resolution.

A drawing plane is described as having infinitely fine spatial, color, and opacity resolution, and as allowing coordinates of unbounded positive or negative magnitude. A viewport into a drawing plane, on the other hand, views only a finite region (usually rectangular) of the drawing plane. Furthermore, a viewport has limited spatial resolution and can only produce a limited number of colors. These limitations are imposed by the display hardware on which the viewport is displayed. A viewport also has limited opacity resolution, determined by the finite arithmetic used in the drawing engine (which may be hardware or software or both).

Coordinates are real numbers in implementation-defined units. Often these units equal the spatial resolution of a viewport, so that a line of thickness 1 is equivalent to the thinnest visible line. However, this equivalence is not required and should not be assumed by application programs.

A valid CLIM implementation can be quite restrictive in the size and resolution of its viewports. For example, the spatial resolution might be only a few dozen points per inch, the maximum size might be only a few hundred points on a side, and there could be as few as two displayable colors (usually black and white). The fully transparent and fully opaque opacity levels must always be supported, but a valid CLIM implementation might support only a few opacity levels in between (or possibly even none). A valid CLIM implementation might implement color blending and unsaturated colors by stippling, although it is preferred, when possible, for a viewport to display a uniform color as a uniform color rather than as a perceptible stipple.

When CLIM records the output to a sheet, there are no such limitations since CLIM just remembers the drawing operations that were performed, not the results of rendering.

CLIM provides some ways to ask what resolution limits are in effect for a medium. See Chapter 10 for their descriptions.

The application programmer uses the CLIM graphic drawing model as an interface to describe the intended visual appearance. An implementation does its best to approximate that ideal appearance in a viewport, within its limitations of spatial resolution, color resolution, number of simultaneously displayable colors, and drawing speed. This will usually require tradeoffs, for example between speed and accuracy, and each implementation must make these tradeoffs according to its own hardware/software environment and user concerns. For example, if the actual device supports a limited number of colors, the desired color may be approximated by techniques such as dithering or stippling. If the actual device cannot draw curves exactly, they may be approximated, with or without anti-aliasing. If the actual device has limited opacity resolution, color blending may be approximate. A viewport might display colors that don't appear in the drawing plane, both because of color and opacity approximation and because of anti-aliasing at the edges of drawn shapes.

It is likely that different implementations will produce somewhat different visual appearance when running the same application. If an application requires more detailed control, it must resort to a lower-level interface, and will become less portable as a result. These lower-level interfaces will be documented on a per-platform basis.

Drawing computations are always carried out "in color", even if the viewport is only capable of displaying black and white. In other words, the CLIM drawing model is always the fully general model, even if an implementation's color resolution is limited enough that full use of the model is not possible. Of course an application that fundamentally depends on color will not work well on a viewport that cannot display color. Other applications will degrade gracefully.

Whether the implementation uses raster graphics or some other display technique is invisible at this interface. CLIM does not specify the existence of pixels nor the exact details of scan conversion, which will vary from one drawing engine to the next.

Performance will also vary between implementations. This interface is defined in terms of simple conceptual operations, however an actual implementation may use caching, specialized object representations, and other optimizations to avoid materializing storage-intensive or computation-costly intermediate results and to take advantage of available hardware.

12.4 Rendering Conventions for Geometric Shapes

The intent of this section is to describe the conventions for how CLIM should render a shape on a display device. These conventions and the accompanying examples are meant to describe a set of goals that a CLIM implementation should try to meet. However, compliant CLIM implementations may deviate from these goals if necessary (for example, if the rendering performance on a specific platform would be unacceptably slow if these goals were met exactly and implementors feel that users would be better served by speed than by accuracy). Note that we discuss only pixel-based display devices here, which are the most common, but by no means the only, sort of display device that can be supported by CLIM.

When CLIM draws a geometric shape on some sort of display device, the idealized geometric shape must somehow be rendered on the display device. The geometric shapes are made up of a set of mathematical points, which have no size; the rendering of the shape is usually composed of pixels, which are roughly square. These pixels exist in "device coordinates", which are gotten by transforming the user-supplied coordinates by all of the user-supplied transformation, the medium transformation, and the transformation that maps from the sheet to the display device. (Note that if the last transformation is a pure translation that translates by an integer multiple of device units, then it has no effect on the rendering other than placement of the figure drawn on the display device.)

Roughly speaking, a pixel is affected by drawing a shape only when it is inside the shape (we will define what we mean by "inside" in a moment). Since pixels are little squares and the abstract points have no size, for most shapes there will be many pixels that lie only partially inside the shape. Therefore, it is important to describe the conventions used by CLIM as to which pixels should be affected when drawing a shape, so that the proper interface to the per-platform rendering engine can be constructed. (It is worth noting that on devices that support color or grayscale, the rendering engine may attempt to draw a pixel that is partially inside the shape darker or lighter, depending on how much of it is inside the shape. This is called *anti-aliasing*.) The conventions used by CLIM is the same as the conventions used by X11:

- A pixel is a addressed by its upper-left corner.
- A pixel is considered to be *inside* a shape, and hence affected by the rendering of that shape, if the center of the pixel is inside the shape. If the center of the pixel lies exactly on the boundary of the shape, it is considered to be inside if the inside of the shape is immediately to the right (increasing x direction on the display device) of the center point of the pixel. If the center of the pixel lies exactly on a horizontal boundary, it is considered to be inside if the inside of the shape is immediately below (increasing y direction on the display device) the center point of the pixel.
- An unfilled shape is drawn by taking the filled shape consisting of those points that are within 1/2 the line thickness from the outline curve (using a normal distance function, that

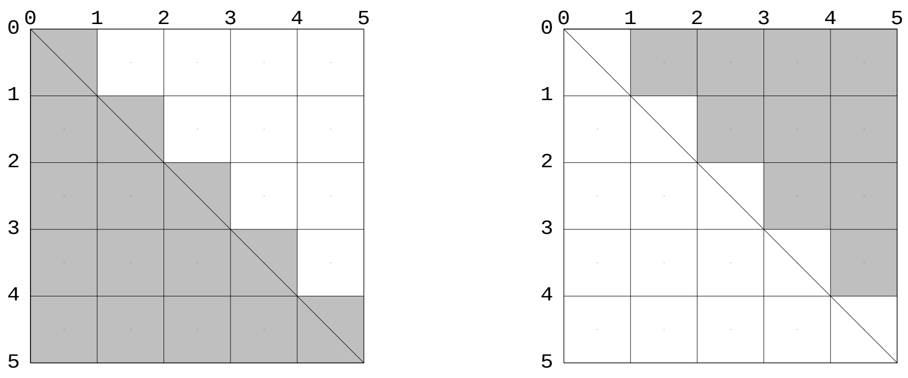


Figure 12.1: Pixel assignment with boundary on decision points.

is, the length of the line drawn at right angles to the tangent to the outline curve at the nearest point), and applying the second rule, above.

It is important to note that these rules imply that the decision point used for insideness checking is offset from the point used for addressing the pixel by half a device unit in both the x and y directions. It is worth considering the motivations for these conventions.

When two shapes share a common edge, it is important that only one of the shapes own any pixel. The two triangles in Figure 12.1 illustrate this. The pixels along the diagonal belong to the lower figure. When the decision point of the pixel (its center) lies to one side of the line or the other, there is no issue. When the boundary passes through a decision point, which side the inside of the figure is on is used to decide. These are the triangles that CLIM implementations should attempt to draw in this case.

The reason for choosing the decision point half a pixel offset from the address point is to reduce the number of common figures (such as rectilinear lines and rectangles with integral coordinates) that invoke the boundary condition rule. This usually leads to more symmetrical results. For instance, in Figure 12.2, we see a circle drawn when the decision point is the same as the address point. The four lighter points are indeterminate: it is not clear whether they are inside or outside the shape. Since we want to have each boundary case determined according to which side has the figure on it, and since we must apply the same rule uniformly for all figures, we have no choice but to pick only two of the four points, leading to an undesirable lopsided figure.

If we had instead chosen to take all four boundary points, we would have a nice symmetrical figure. However, since this figure is symmetrical about a whole pixel, it is one pixel wider than it ought to be. The problem with this can be seen clearly in Figure 12.3 if we attempt to draw a rectangle and circle overlaid with the following code:

```
(defun draw-test (medium radius)
  (draw-circle* medium 0 0 radius :ink +foreground-ink+)
  (draw-rectangle* medium (- radius) (- radius) (+ radius) (+ radius)
                   :ink +flipping-ink+))
```

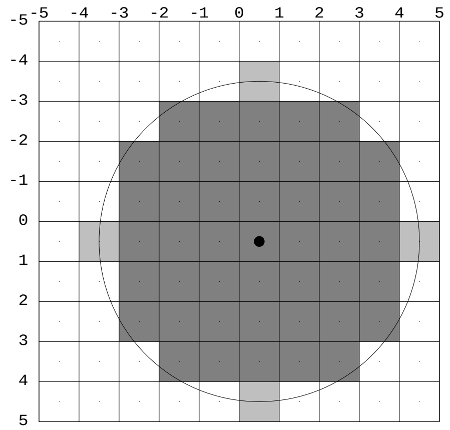


Figure 12.2: Choosing any two of the shaded pixels causes asymmetry.

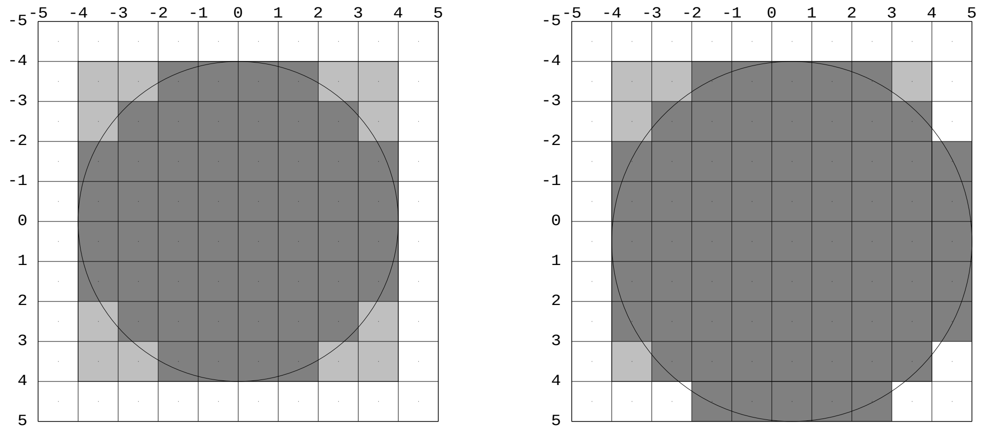


Figure 12.3: Two forms of a circle inscribed in a rectangle.

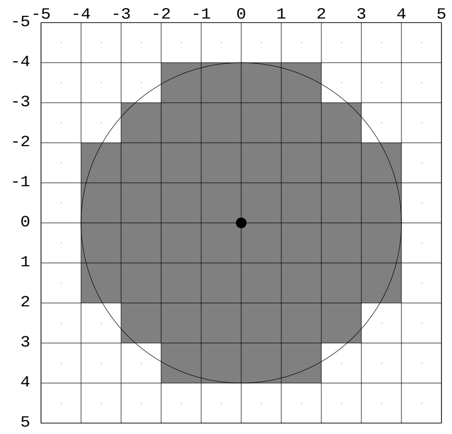


Figure 12.4: An aesthetically pleasing circle.

It is for this reason that we choose to have the decision point at the center of the pixel. This draws circles that look like the one in Figure 12.4. It is this shape that CLIM implementations should attempt to draw.

A consequence of these rendering conventions is that, when the start or end coordinate (minus 1/2 the line thickness, if the shape is a path) is not an integer, then rendering is not symmetric under reflection transformations. Thus to correctly and portably draw an outline of thickness 1 around a (rectilinear) rectangular area with integral coordinates, the outline path must have half-integral coordinates. Drawing rectilinear areas whose boundaries are not on pixel boundaries cannot be guaranteed to be portable. Another way to say the same thing is that the "control points" for a rectangular area are at the corners, while the control points for a rectilinear path are in the center of the path, not at the corners. Therefore, in order for a path and an area to abut seamlessly, the coordinates of the path must be offset from the coordinates of the area by half the path's thickness.

12.4.1 Permissible Alternatives During Rendering

Some platforms may distinguish between lines of the minimum thinness from lines that are thicker than that. The two rasterizations depicted in Figure 12.5 are both perfectly reasonable rasterizations of tilted lines that are a single device unit wide. The right-hand line is drawn as a tilted rectangle, the left as the "thinnest visible" line.

For thick lines, a platform may choose to draw the exact tilted fractional rectangle, or the coordinates of that rectangle might be rounded so that it is distorted into another polygonal shape. The latter case may be prove to be faster on some platforms. The two rasterizations depicted in Figure 12.6 are both reasonable.

The decision about which side of the shape to take when a boundary line passes through the decision point is made arbitrarily, although we have chosen to be compatible with the X11 definition. This is not necessarily the most convenient decision. The main problem with this is illustrated by the case of a horizontal line (see Figure 12.7). Our definition chooses to draw

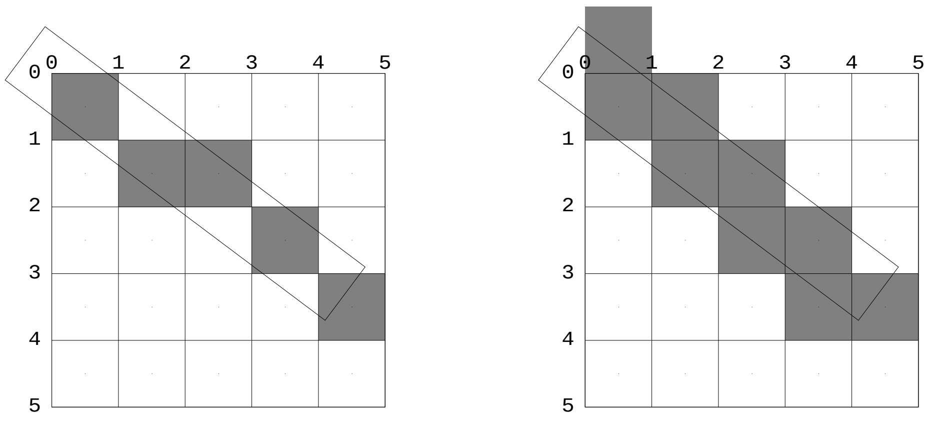


Figure 12.5: Two examples of lines of thickness 1.

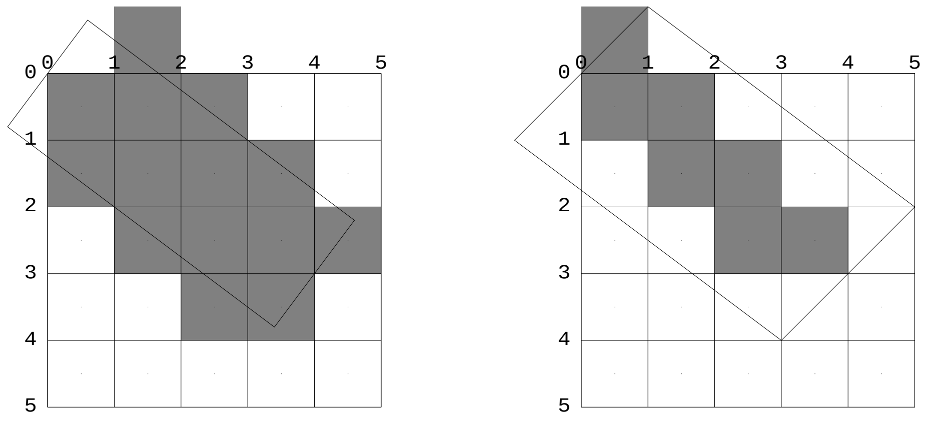


Figure 12.6: Two examples of lines of thickness 2.

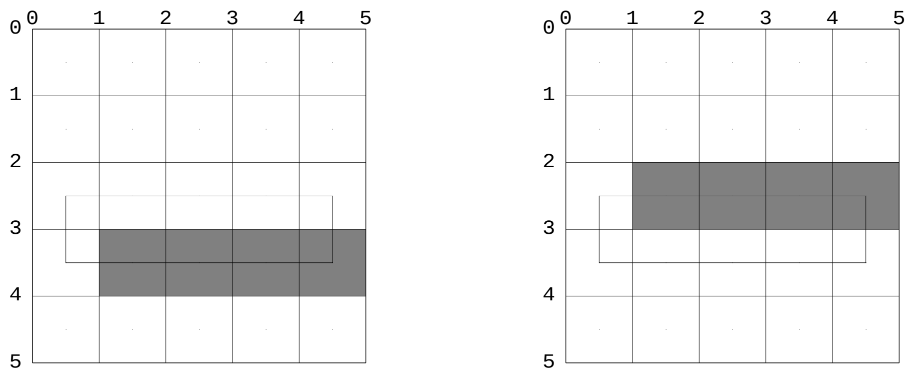


Figure 12.7: Two possible definitions of horizontal lines. Left figure is X11 definition.

the rectangular slice above the coordinates, since those pixels are the ones whose centers have the figure immediately above them. This definition makes it simpler to draw rectilinear borders around rectilinear areas.

12.5 Drawing Functions

Each drawing function takes keyword arguments allowing any drawing option or suboption to be supplied separately in the call to the function. In some implementations of CLIM, the drawing functions may ignore drawing options that are irrelevant to that function; in other implementations, an error may be signalled. See Chapter 10 for a more complete discussion of the drawing options. An error will be signalled if any drawing function is called on a sheet that is mute for output.

While the functions in this section are specified to be called on mediums, they can also be called on sheets and streams. CLIM implementations will typically implement the method on a medium, and write a "trampoline" on the various sheet and stream classes that trampolines to the medium.

Implementation note: The drawing functions are all specified as ordinary functions, not as generic functions. This is intended to ease the task of writing compile-time optimizations that avoid keyword argument taking, check for such things as constant drawing options, and so forth. If you need to specialize any of the drawing methods, use `define-graphics-method`.

Each drawing function comes in two forms, a "structured" version and a "spread" version. The structured version passes points, whereas the spread version passes coordinates. See Section 2.3 for more information on this.

Any drawing functions may create an *output record* that corresponds to the figure being drawn. See Chapter 15 for a complete discussion of output recording. During output recording, none of these functions capture any arguments that are points, point sequences, coordinate sequences, or text strings. Line styles, text styles, transformations, and clipping regions may be captured.

Note that the CLIM specification does not specify more complex shapes such as cubic splines and Bézier curves. These are suitable candidates for extensions to CLIM.

12.5.1 Basic Drawing Functions

⇒ `draw-point` *medium point* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]
⇒ `draw-point*` *medium x y* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]

These functions (structured and spread arguments, respectively) draw a single point on the *medium medium* at the *point point* (or the position (x, y)).

The unit and thickness components of the current line style (see Chapter 10) affect the drawing of the point by controlling the size on the display device of the "blob" that is used to render the point.

⇒ `draw-points` *medium points* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]
⇒ `draw-points*` *medium position-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit* [*Function*]

These functions (structured and spread arguments, respectively) draw a set of points on the *medium medium*. *points* is a sequence of point objects; *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *position-seq* does not contain an even number of elements.

Ignoring the drawing options, these functions exist as equivalents to

```
(map nil #'(lambda (point) (draw-point medium point)) points)
```

and

```
(do ((i 0 (+ i 2)))
    ((= i (length position-seq)))
  (draw-point* medium (elt position-seq i) (elt position-seq (+ i 1))))
```

for convenience and efficiency.

⇒ `draw-line` *medium point1 point2* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-line*` *medium x1 y1 x2 y2* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a line segment on the *medium medium* from the *point point1* to *point2* (or from the position (*x1*,*y1*) to (*x2*,*y2*)).

The current line style (see Chapter 10) affects the drawing of the line in the obvious way, except that the joint shape has no effect. Dashed lines start dashing at *point1*.

⇒ `draw-lines` *medium points* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-lines*` *medium position-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a set of disconnected line segments. *points* is a sequence of point objects; *position-seq* is a sequence of coordinate pairs. It is an error if *position-seq* does not contain an even number of elements.

Ignoring the drawing options, these functions are equivalent to

```
(do ((i 0 (+ i 2)))
    ((= i (length points)))
  (draw-line medium (elt points i) (elt points (1+ i))))
```

and

```
(do ((i 0 (+ i 4)))
    ((= i (length position-seq)))
  (draw-line* medium (elt position-seq i)       (elt position-seq (+ i 1))
                     (elt position-seq (+ i 2)) (elt position-seq (+ i 3))))
```

⇒ `draw-polygon` *medium point-seq* `&key` *(filled* `t`*) (closed* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape* [*Function*]
⇒ `draw-polygon*` *medium coord-seq* `&key` *(filled* `t`*) (closed* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape* [*Function*]

Draws a polygon or polyline on the medium *medium*. When *filled* is *false*, this draws a set of connected lines, otherwise it draws a filled polygon. If *closed* is *true* (the default) and *filled* is *false*, it ensures that a segment is drawn that connects the ending point of the last segment to the starting point of the first segment. The current line style (see Chapter 10) affects the drawing of unfilled polygons in the obvious way. The cap shape affects only the "open" vertices in the case when *closed* is *false*. Dashed lines start dashing at the starting point of the first segment, and may or may not continue dashing across vertices, depending on the window system.

point-seq is a sequence of point objects; *coord-seq* is a sequence of coordinate pairs. It is an error if *coord-seq* does not contain an even number of elements.

If *filled* is *true*, a closed polygon is drawn and filled in. In this case, *closed* is assumed to be *true* as well.

⇒ `draw-rectangle` *medium point1 point2* `&key` *(filled* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape* [*Function*]

⇒ `draw-rectangle*` *medium x1 y1 x2 y2* `&key` *(filled* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape* [*Function*]

Draws either a filled or unfilled rectangle on the *medium medium* that has its sides aligned with the coordinate axes of the native coordinate system. One corner of the rectangle is at the position (*x1*,*y1*) and the opposite corner is at (*x2*,*y2*). The arguments *x1*, *y1*, *x2*, and *y1* are real numbers that are canonicalized in the same way as for `make-bounding-rectangle`. *filled* is as for `draw-polygon*`.

The current line style (see Chapter 10) affects the drawing of unfilled rectangles in the obvious way, except that the cap shape has no effect.

⇒ `draw-rectangles` *medium points* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape* [*Function*]
⇒ `draw-rectangles*` *medium position-seq* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a set of rectangles. *points* is a sequence of point objects; *position-seq* is a sequence of coordinate pairs. It is an error if *position-seq* does not contain an even number of elements.

Ignoring the drawing options, these functions are equivalent to

```
(do ((i 0 (+ i 2)))
    ((= i (length points)))
  (draw-rectangle medium (elt points i) (elt points (1+ i))))
```

and

```
(do ((i 0 (+ i 4)))
    ((= i (length position-seq)))
  (draw-rectangle* medium (elt position-seq i)       (elt position-seq (+ i 1))
                          (elt position-seq (+ i 2)) (elt position-seq (+ i 3))))
```

⇒ `draw-ellipse` *medium center-pt radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-ellipse*` *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw an ellipse (when *filled* is *true*, the default) or an elliptical arc (when *filled* is *false*) on the *medium medium*. The center of the ellipse is the *point center-pt* (or the position (*center-x*,*center-y*)).

Two vectors, (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*) specify the bounding parallelogram of the ellipse as explained in Chapter 3. All of the radii are real numbers. If the two vectors are collinear, the ellipse is not well-defined and the `ellipse-not-well-defined` error will be signalled. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. Angles are measured with respect to the positive x axis. The elliptical arc runs positively (counter-clockwise) from *start-angle* to *end-angle*. The default for *start-angle* is 0; the default for *end-angle* is 2π.

In the case of a "filled arc" (that is when *filled* is *true* and *start-angle* or *end-angle* are supplied and are not 0 and 2π), the figure drawn is the "pie slice" area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from *start-angle* to *end-angle*.

When drawing unfilled ellipses, the current line style (see Chapter 10) affects the drawing in the obvious way, except that the joint shape has no effect. Dashed elliptical arcs start "dashing" at *start-angle*.

⇒ `draw-circle` *medium center-pt radius* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-circle*` *medium center-x center-y radius* `&key` *(filled* `t`*) start-angle end-angle ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a circle (when *filled* is *true*, the default) or a circular arc (when *filled* is *false*) on the medium *medium*. The center of the circle is *center-pt* or (*center-x*,*center-y*) and the radius is *radius*. These are just special cases of `draw-ellipse` and `draw-ellipse*`. *filled* is as for `draw-ellipse*`.

start-angle and *end-angle* allow the specification of an arc rather than a complete circle in the same manner as that of the ellipse functions, above.

The "filled arc" behavior is the same as that of an ellipse, above.

⇒ `draw-text` *medium string-or-char point* `&key` *text-style (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*) toward-point transform-glyphs ink clipping-region transformation text-style text-family text-face text-size* [*Function*]
⇒ `draw-text*` *medium string-or-char x y* `&key` *text-style (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*) toward-x toward-y transform-glyphs ink clipping-region transformation text-style text-family text-face text-size* [*Function*]

The text specified by *string-or-char* is drawn on the medium *medium* starting at the position specified by the point *point* (or the position (x, y)). The exact definition of "starting at" is dependent on *align-x* and *align-y*. *align-x* is one of `:left`, `:center`, or `:right`. *align-y* is one of `:baseline`, `:top`, `:center`, or `:bottom`. *align-x* defaults to `:left` and *align-y* defaults to `:baseline`; with these defaults, the first glyph is drawn with its left edge and its baseline at *point*.

text-style defaults to `nil`, meaning that the text will be drawn using the current text style of the medium.

start and *end* specify the start and end of the string, in the case where *string-or-char* is a string. If *start* is supplied, it must be an integer that is less than the length of the string. If *end* is supplied, it must be an integer that is less than the length of the string, but greater than or equal to *start*.

Normally, glyphs are drawn from left to right no matter what transformation is in effect. *toward-x* or *toward-y* (derived from *toward-point* in the case of `draw-text`) can be used to change the direction from one glyph to the next one. For example, if *toward-x* is less than the *x* position of *point*, then the glyphs will be drawn from right to left. If *toward-y* is greater than the *y* position of *point*, then the glyphs' baselines will be positioned one above another. More precisely, the reference point in each glyph lies on a line from *point* to *toward-point*, and the spacing of each glyph is determined by packing rectangles along that line, where each rectangle is "char-width" wide and "char-height" high.

If *transform-glyphs* is *true*, then each glyph is transformed as an image before it is drawn. That is, if a rotation transformation is in effect, then each glyph will be rotated individually. If *transform-glyphs* is not supplied, then the individual glyphs are not subject to the current transformation. It is permissible for CLIM implementations to ignore *transform-glyphs* if it is too expensive to implement.

12.5.2 Compound Drawing Functions

CLIM also provides a few compound drawing functions. The compound drawing functions could be composed by a programmer from the basic drawing functions, but are provided by CLIM because they are commonly used.

⇒ `draw-arrow` *medium point-1 point-2* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shapeto-head from-head head-length head-width* [*Function*]
⇒ `draw-arrow*` *medium x1 y1 x2 y2* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shapefrom-head to-head head-length head-width* [*Function*]

These functions (structured and spread arguments, respectively) draw a line segment on the *medium medium* from the *point point1* to *point2* (or from the position (*x1*,*y1*) to (*x2*,*y2*)). If *to-head* is *true* (the default), then the "to" end of the line is capped by an arrowhead. If *from-head* is *true* (the default is *false*), then the "from" end of the line is capped by an arrowhead. The arrowhead has length *head-length* (default 10) and width *head-width* (default 5).

The current line style (see Chapter 10) affects the drawing of the line portion of the arrow in the obvious way, except that the joint shape has no effect. Dashed arrows start dashing at *point1*.

⇒ `draw-oval` *medium center-pt x-radius y-radius* `&key` *(filled* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]
⇒ `draw-oval*` *medium center-x center-y x-radius y-radius* `&key` *(filled* `t`*) ink clipping-region transformation line-style line-thickness line-unit line-dashes line-cap-shape* [*Function*]

These functions (structured and spread arguments, respectively) draw a filled or unfilled oval (that is, a "race-track" shape) on the *medium medium*. The oval is centered on *center-pt* (or (*center-x*,*center-y*)). If *x-radius* or *y-radius* is 0, then a circle is drawn with the specified non-zero radius. Other, a figure is drawn that results from drawing a rectangle with dimension x-radius by *y-radius*, and the replacing the two short sides with a semicircular arc of the appropriate size.

12.6 Pixmaps

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

The exact representation of a pixmap is explicitly unspecified. There is no interaction between the pixmap operations and output recording, that is, displaying a pixmap on a sheet or medium is a pure drawing operation that affects only the display, not the output history. Some mediums may not support pixmaps; in this case, an error will be signalled.

⇒ `allocate-pixmap` *medium width height* [*Generic Function*]

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

The resulting pixmap will be at least *width* units wide, *height* units high, and as deep as is necessary to store the information for the medium. The exact representation of pixmaps is explicitly unspecified.

The returned value is the pixmap.

⇒ `deallocate-pixmap` *pixmap* [*Generic Function*]

Deallocates the pixmap *pixmap*.

⇒ `pixmap-width` *pixmap* [*Generic Function*]
⇒ `pixmap-height` *pixmap* [*Generic Function*]
⇒ `pixmap-depth` *pixmap* [*Generic Function*]

These functions return, respectively, the width, height, and depth of the pixmap *pixmap*. These values may be different from the programmer-specified values, since some window systems need to allocate pixmaps only of particular sizes.

⇒ `copy-to-pixmap` *medium medium-x medium-y width height* `&optional` *pixmap (pixmap-x* `0`*)* *(pixmap-y* `0`*)* [*Function*]

Copies the pixels from the medium *medium* starting at the position specified by (*medium-*

x,*medium-y*) into the pixmap *pixmap* at the position specified by (*pixmap-x*,*pixmap-y*). A rectangle whose width and height is specified by *width* and *height* is copied. *medium-x* and *medium-y* are specified in user coordinates. (If *medium* is a medium or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.) The copying must be done by `medium-copy-copy`.

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

The returned value is the pixmap.

⇒ `copy-from-pixmap` *pixmap pixmap-x pixmap-y width height medium medium-x medium-y* [*Function*]

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

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

The returned value is the pixmap.

⇒ `copy-area` *medium from-x from-y width height to-x to-y* [*Generic Function*]

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

⇒ `medium-copy-area` *from-drawable from-x from-y width height to-drawable to-x to-y* [*Generic Function*]

Copies the pixels from the source drawable *from-drawable* at the position (*from-x*,*from-y*) to the destination drawable *to-drawable* at the position (*to-x*,*to-y*). A rectangle whose width and height is specified by *width* and *height* is copied. *from-x*, *from-y*, *to-x*, and *to-y* are specified in user coordinates. The *x* and *y* are transformed by the user transformation.

This is intended to specialize on both the *from-drawable* and *to-drawable* arguments. *from-drawable* and *to-drawable* may be either mediums or pixmaps.

⇒ `with-output-to-pixmap` *(medium-var medium* `&key` *width height)* `&body` *body* [*Macro*]

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

width and *height* are integers that give the width and height of the pixmap. If they are unsupplied, the result pixmap will be large enough to contain all of the output done by *body*.

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

The returned value is a pixmap that can be drawn onto *medium* using `copy-from-pixmap`.

12.7 Graphics Protocols

Every medium must implement methods for the various graphical drawing generic functions. Furthermore, every sheet that supports the standard output protocol must implement these methods as well; often, the sheet methods will trampoline to the methods on the sheet's medium. All of these generic functions take the same arguments as the non-generic spread function equivalents, except the arguments that are keyword arguments in the non-generic functions are positional arguments in the generic functions.

Every medium must implement methods for the various graphical drawing generic functions. All of these generic functions take as (specialized) arguments the medium, followed by the drawing function-specific arguments, followed by the ink, line style (or text style), and clipping region.

The drawing function-specific arguments will either be x and y positions, or a sequence of x and y positions. These positions will be in medium coordinates, and must be transformed by applying the medium's device transformation in order to produce device coordinates. Note that the user transformation will have already been applied to the positions when the medium-specific drawing function is called. However, the medium-specific drawing function will still need to apply the device transformation to the positions in order to draw the graphics in the appopriate place on the host window.

The ink, line style, text style, and clipping regions arguments are not part of the medium-specific drawing functions. They must be extracted from the medium object. Each medium-specific method will decode the ink, line (or text) style, and clipping region in a port-specific way and communicate it to the underlying port.

12.7.1 General Behavior of Drawing Functions

Using `draw-line*` as an example, calling any of the drawing functions specified above results in the following series of function calls on a non-output recording sheet:

- A program calls `draw-line*` on either a sheet or a medium, *x1*, *y1*, *x2*, and *y2*, and perhaps some drawing options.
- `draw-line*` merges the supplied drawing options into the medium, and then calls `medium-draw-line*` on the sheet or medium. (Note that a compiler macro could detect the case

where there are no drawing options or constant drawing options, and do this at compile time.)

- If `draw-line*` was called on a sheet, the `medium-draw-line*` on the sheet trampolines to the medium's `medium-draw-line*` method.
- An `:around` method for `medium-draw-line*` performs the necessary user transformations by applying the medium transformation to *x1*, *y1*, *x2*, and *y2*, and to the clipping region, and then calls the medium-specific method.
- The "real" `medium-draw-line*` transforms the start and end coordinates of the line by the sheet's device transformation, decodes the ink and line style into port-specific objects, and finally invokes a port-specific function (such as `xlib:draw-line`) to do the actual drawing.

12.7.2 Medium-specific Drawing Functions

All mediums and all sheets that support the standard output protocol must implement methods for the following generic functions.

The method for each of these drawing functions on the most specific, implementation-dependent medium class will transform the coordinates by the device transformation of the medium's sheet, extract the medium's port-specific "drawable", and then invoke a port-specific drawing function (such as `xlib:draw-line`) to do the actual drawing.

An `:around` on `basic-medium` for each of the drawing functions will have already transformed the user coordinates to medium coordinates before the most specific, implementation-dependent method is called.

Implementation note: CLIM implementations should provide "trampoline" methods on sheets that support the standard output protocol that simply call the same generic function on the medium. Sheets that support output recording will require extra mechanism before delegating to the medium in order to implement such functionality as creating output records and handling scrolling.

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

Draws a point on the medium *medium*.

⇒ `medium-draw-points*` *medium coord-seq* [*Generic Function*]

Draws a set of points on the medium *medium*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements.

⇒ `medium-draw-line*` *medium x1 y1 x2 y2* [*Generic Function*]

Draws a line on the medium *medium*. The line is drawn from $(x1, y1)$ to $(x2, y2)$.

⇒ `medium-draw-lines*` *stream position-seq* [*Generic Function*]

Draws a set of disconnected lines on the medium *medium*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. Each successive pair of coordinate pairs is taken as the start and end position of each line. It is an error if *coord-seq* does not contain an even number of elements.

⇒ `medium-draw-polygon*` *medium coord-seq closed* [*Generic Function*]

Draws a polygon or polyline on the medium *medium*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. Each successive coordinate pair is taken as the position of one vertex of the polygon.

⇒ `medium-draw-rectangle*` *medium x1 y1 x2 y2* [*Generic Function*]

Draws a rectangle on the medium *medium*. The corners of the rectangle are at $(x1, y1)$ and $(x2, y2)$.

⇒ `medium-draw-rectangles*` *medium coord-seq* [*Generic Function*]

Draws a set of rectangles on the medium *medium*. *coord-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *coord-seq* does not contain an even number of elements. Each successive pair of coordinate pairs is taken as the upper-left and lower-right corner of the rectangle.

⇒ `medium-draw-ellipse*` *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle* [*Generic Function*]

Draws an ellipse or elliptical arc on the medium *medium*. The center of the ellipse is at (x, y), and the radii are specified by the two vectors (*radius-1-dx*,*radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*).

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. Note that the medium and device transformations must be applied to the angles as well.

⇒ `medium-draw-text*` *medium text x y (start* `0`*) end (align-x* `:left`*) (align-y* `:baseline`*) toward-x toward-y transform-glyphs* [*Generic Function*]

Draws a character or a string on the medium *medium*. The text is drawn starting at (x, y), and towards (*toward-x*,*toward-y*). In Some implementations of CLIM, `medium-draw-text*` may call either `medium-draw-string*` or `medium-draw-character*` in order to draw the text.

12.7.3 Other Medium-specific Output Functions

⇒ `medium-finish-output` *medium* [*Generic Function*]

Ensures that all the output sent to *medium* has reached its destination, and only then return *false*. This is used by `finish-output`.

⇒ `medium-force-output` *medium* [*Generic Function*]

Like `medium-finish-output`, except that it may return *false* without waiting for the output to complete. This is used by `force-output`.

⇒ `medium-clear-area` *medium left top right bottom* [*Generic Function*]

Clears an area on the medium *medium* by filling the rectangle whose edges are at *left*, *top*, *right*, and *bottom* with the medium's background ink. *left*, *top*, *right*, and *bottom* are in thed medium's coordinate system.

The default method on `basic-medium` simply uses `draw-rectangle*` to clear the area. Some host window systems has special functions that are faster than `draw-rectangle*`.

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

Causes an audible sound to be played on the medium. The default method does nothing.

Chapter 13

Drawing in Color

This chapter describes the `:ink` drawing option and the simpler values that can be supplied for that option, such as colors. More complex values that have a regular or irregular pattern in the ink are described in Chapter 14.

Major issue: *We need to add a thing called a "palette", which is simply an abstract color map. Palettes are primarily used as a resource for the limited number of colors on most hosts. Do we need to be able to used them to more directly control color maps, to do "color map animation", for example? — SWM*

Minor issue: *We need to add a thing called a "raster ink", which includes things like plane masks, pixel values, etc. Be clear that this is platform and device dependent. — SWM*

13.1 The `:ink` Drawing Option

The `:ink` drawing option, used with the drawing functions described in Chapter 12, can take as its value:

- a color,
- an opacity or the constant `+transparent-ink+`,
- the constant `+foreground-ink+`,
- the constant `+background-ink+`,
- a flipping ink, or
- other values described in Chapter 14

More exactly, an ink can be any member of the class `design`. For now you may think of a design as a possibly translucent color. More general designs are described in Chapter 14.

The drawing functions work by selecting a region of the drawing plane and painting it with color. The region to be painted is the intersection of the shape specified by the drawing function and the `:clipping-region` drawing option, which is then transformed by the `:transformation` drawing option. The `:ink` drawing option is a design that specifies a new arrangement of colors (and opacities) in this region of the medium's drawing plane. Any viewports or dataports attached to this drawing plane are updated accordingly. The `:ink` drawing option is never affected by the `:transformation` drawing option nor by the medium's transformation; this ensures that stipple patterns on adjacent sheets join seamlessly.

Minor issue: *The description of how the clipping region and transformations contribute isn't good enough. It is true if there are no other transformations and clipping regions present, and both are specified in the current drawing operation. But it doesn't say what happens if things are nested. I'm not sure it needs to. Rather, I think it should just say that the the region is clipped by the current clipping region in effect, then transformed by the current transform in effect, and that the rules for these are discussed in the drawing options section. — DCPL*

Drawing consists conceptually of the following sequence of operations, performed in parallel at every point in the drawing plane. Of course, the actual implementation does not involve an infinite (or large parallel) computation.

1. The design specifies a color and an opacity at the point. These can depend on the drawing plane's current color and opacity, on the medium's foreground color, and on the medium's background color.
2. The color blending function is applied to the design's color and opacity and the drawing plane's color and opacity, returning a new color and opacity for the point.
3. The drawing plane's color and opacity at that point are set to the new color and opacity.

13.2 Basic Designs

⇒ `design` *[Protocol Class]*

A design is an object that represents a way of arranging colors and opacities in the drawing plane. The `design` class is the protocol class for designs. If you want to create a new class that behaves like a design, it should be a subclass of `design`. All instantiable subclasses of `design` must obey the design protocol.

The fundamental operation of the CLIM graphic drawing model is to draw a design onto a drawing plane, thus drawing is always controlled by designs. The designs discussed in this chapter do the same thing at each point in the drawing plane. Chapter 14 discusses more general designs and reveals that regions are also designs.

A design can be characterized in several different ways:

All designs are either *bounded* or *unbounded*. Bounded designs are transparent everywhere beyond a certain distance from a certain point. Drawing a bounded design has no effect on the drawing plane outside that distance. Unbounded designs have points of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.

All designs are either *uniform* or *non-uniform*. Uniform designs have the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.

All designs are either *solid* or *translucent*. At each point a solid design is either completely opaque or completely transparent. A solid design can be opaque at some points and transparent at others. In translucent designs, at least one point has an opacity that is intermediate between completely opaque and transparent.

All designs are either *colorless* or *colored*. Drawing a colorless design uses a default color specified by the medium's foreground design. This is done by drawing with `(compose-in +foreground-ink+` the-colorless-design`)`. See Chapter 14 for the details of `compose-in`.

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

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

13.3 Color

⇒ `color` [*Protocol Class*]

A member of the class `color` is a completely opaque design that represents the intuitive definition of color: white, black, red, pale yellow, and so forth. The visual appearance of a single point is completely described by its color. Drawing a color sets the color of every point in the drawing plane to that color, and sets the opacity to 1. The `color` class is the protocol class for a color, and is a subclass of `design`. If you want to create a new class that behaves like a color, it should be a subclass of `color`. All instantiable subclasses of `color` must obey the color protocol.

All of the standard instantiable color classes provided by CLIM are immutable.

A color can be specified by three real numbers between 0 and 1 (inclusive), giving the amounts of red, green, and blue. Three 0's mean black; three 1's mean white. The intensity-hue-saturation color model is also supported, but the red-green-blue color model is the primary model we will use in the specification.

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

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

The following functions create colors. These functions produce objects that have equivalent effects; the only difference is in how the color components are specified. The resulting objects are indistinguishable when drawn. Whether these functions use the specified values exactly or approximate them because of limited color resolution is unspecified. Whether these functions create a new object or return an existing object with equivalent color component values is unspecified.

⇒ `make-rgb-color` *red green blue* [*Function*]

Returns a member of class `color`. The *red*, *green*, and *blue* arguments are real numbers between 0 and 1 (inclusive) that specify the values of the corresponding color components.

When all three color components are 1, the resulting color is white. When all three color components are 0, the resulting color is black.

⇒ `make-ihs-color` *intensity hue saturation* [*Function*]

Returns a member of class `color`. The *intensity* argument is a real number between 0 and $\sqrt{3}$ (inclusive). The *hue* and *saturation* arguments are real numbers between 0 and 1 (inclusive).

⇒ `make-gray-color` *luminance* [*Function*]

Returns a member of class `color`. *luminance* is a real number between 0 and 1 (inclusive). On a black-on-white display device, 0 means black, 1 means white, and the values in between are shades of gray. On a white-on-black display device, 0 means white, 1 means black, and the values in between are shades of gray.

The following two functions comprise the color protocol. Both of them return the components of a color. All subclasses of `color` must implement methods for these generic functions.

⇒ `color-rgb` *color* [*Generic Function*]

Returns three values, the *red*, *green*, and *blue* components of the *color color*. The values are real numbers between 0 and 1 (inclusive).

⇒ `color-ihs` *color* [*Generic Function*]

Returns three values, the *intensity*, *hue*, and *saturation* components of the *color color*. The first value is a real number between 0 andn $\sqrt{3}$ (inclusive). The second and third values are real numbers between 0 and 1 (inclusive).

13.3.1 Standard Color Names and Constants

Table 13.1 lists the commonly provided color names that can be looked up with `find-named-color`. Application programs can define other colors; these are provided because they are commonly used in the X Windows community, not because there is anything special about these particular colors. This table is a subset of the color listed in the file `/X11/R4/mit/rgb/rgb.txt`, from the X11 R4 distribution.

In addition, the following color constants are provided.

⇒ `+red+` [*Constant*]
⇒ `+green+` [*Constant*]
⇒ `+blue+` [*Constant*]
⇒ `+cyan+` [*Constant*]
⇒ `+magenta+` [*Constant*]

`alice-blue`	`antique-white`	`aquamarine`
`azure`	`beige`	`bisque`
`black`	`blanched-almond`	`blue`
`blue-violet`	`brown`	`burlywood`
`cadet-blue`	`chartreuse`	`chocolate`
`coral`	`cornflower-blue`	`cornsilk`
`cyan`	`dark-goldenrod`	`dark-green`
`dark-khaki`	`dark-olive-green`	`dark-orange`
`dark-orchid`	`dark-salmon`	`dark-sea-green`
`dark-slate-blue`	`dark-slate-gray`	`dark-turquoise`
`dark-violet`	`deep-pink`	`deep-sky-blue`
`dim-gray`	`dodger-blue`	`firebrick`
`floral-white`	`forest-green`	`gainsboro`
`ghost-white`	`gold`	`goldenrod`
`gray`	`green`	`green-yellow`
`honeydew`	`hot-pink`	`indian-red`
`ivory`	`khaki`	`lavender`
`lavender-blush`	`lawn-green`	`lemon-chiffon`
`light-blue`	`light-coral`	`light-cyan`
`light-goldenrod`	`light-goldenrod-yellow`	`light-gray`
`light-pink`	`light-salmon`	`light-sea-green`
`light-sky-blue`	`light-slate-blue`	`light-slate-gray`
`light-steel-blue`	`light-yellow`	`lime-green`
`linen`	`magenta`	`maroon`
`medium-aquamarine`	`medium-blue`	`medium-orchid`
`medium-purple`	`medium-sea-green`	`medium-slate-blue`
`medium-spring-green`	`medium-turquoise`	`medium-violet-red`
`midnight-blue`	`mint-cream`	`misty-rose`
`moccasin`	`navajo-white`	`navy-blue`
`old-lace`	`olive-drab`	`orange`
`orange-red`	`orchid`	`pale-goldenrod`
`pale-green`	`pale-turquoise`	`pale-violet-red`
`papaya-whip`	`peach-puff`	`peru`
`pink`	`plum`	`powder-blue`
`purple`	`red`	`rosy-brown`
`royal-blue`	`saddle-brown`	`salmon`
`sandy-brown`	`sea-green`	`seashell`
`sienna`	`sky-blue`	`slate-blue`
`slate-gray`	`snow`	`spring-green`
`steel-blue`	`tan`	`thistle`
`tomato`	`turquoise`	`violet`
`violet-red`	`wheat`	`white`
`white-smoke`	`yellow`	`yellow-green`

Table 13.1: Standard color names.

⇒ `+yellow+` [*Constant*]
⇒ `+black+` [*Constant*]
⇒ `+white+` [*Constant*]

Constants corresponding to the usual definitions of red, green, blue, cyan, magenta, yellow, black, and white.

13.3.2 Contrasting Colors

⇒ `make-contrasting-inks` *n* `&optional` *k* [*Function*]

If *k* is not supplied, this returns a vector of *n* designs with recognizably different appearance. Elements of the vector are guaranteed to be acceptable values for the `:ink` argument to the drawing functions, and will not include `+foreground-ink+`, `+background-ink+`, or `nil`. Their class is otherwise unspecified. The vector is a fresh object that may be modified.

If *k* is supplied, it must be an integer between 0 and $n - 1$ (inclusive), in which case `make-contrasting-inks` returns the *k*'th design rather than returning a vector of designs.

If the implementation does not have *n* different contrasting inks, `make-contrasting-inks` signals an error. This will not happen unless *n* is greater than eight.

The rendering of the design may be a color or a stippled pattern, depending on whether the output medium supports color.

⇒ `contrasting-inks-limit` *port* [*Generic Function*]

Returns the number of contrasting colors (or stipple patterns if *port* is monochrome or grayscale) that can be rendered on any medium on the port *port*. Implementations are encouraged to make this as large as possible, but it must be at least 8. All classes that obey the medium protocol must implement a method for this generic function.

13.4 Opacity

⇒ `opacity` [*Protocol Class*]

A member of the class `opacity` is a completely colorless design that is typically used as the second argument to `compose-in` to adjust the opacity of another design. See Chapter 14 for the details of `compose-in`. The `opacity` class is the protocol class for an opacity, and is a subclass of `design`. If you want to create a new class that behaves like an opacity, it should be a subclass of `opacity`. All instantiable subclasses of `opacity` must obey the opacity protocol.

All of the standard instantiable opacity classes provided by CLIM are immutable.

Opacity controls how graphical output covers previous output. Opacity can vary from totally opaque to totally transparent. Intermediate opacity values result in color blending so that the earlier picture shows through what is drawn on top of it.

An opacity may be specified by a real number between 0 and 1 (inclusive). 0 is completely transparent, 1 is completely opaque, fractions are translucent. The opacity of a design is the degree to which it hides the previous contents of the drawing plane when it is drawn.

The fully transparent and fully opaque opacity levels (that is, opacities 0 and 1) must always be supported, but a valid CLIM implementation might only support a handful of opacity levels in between (including none). A valid CLIM implementation might implement color blending and unsaturated colors by stippling, although it is preferred, when possible, for a viewport to display a uniform color as a uniform color rather than as a perceptible stipple.

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

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

The following function returns an opacity:

⇒ `make-opacity` *value* [*Function*]

Returns a member of class `opacity` whose opacity is *value*, which is a real number in the range from 0 to 1 (inclusive), where 0 is fully transparent and 1 is fully opaque.

⇒ `+transparent-ink+` [*Constant*]

An fully transparent ink, that is, an opacity whose value is 0. This is typically used as the "background" ink in a call to `make-pattern`.

The following function returns the sole component of an opacity. This is the only function in the opacity protocol. All subclasses of `opacity` must implement methods for this generic function.

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

Returns the opacity value of the *opacity opacity*, which is a real number in the range from 0 to 1 (inclusive).

13.5 Color Blending

Drawing a design that is not completely opaque at all points allows the previous contents of the drawing plane to show through. The simplest case is drawing a solid design: where the design is opaque, it replaces the previous contents of the drawing plane; where the design is transparent, it leaves the drawing plane unchanged. In the more general case of drawing a translucent design, the resulting color is a blend of the design's color and the previous color of the drawing plane. For purposes of color blending, the drawn design is called the foreground and the drawing plane is called the background.

The function `compose-over` performs a similar operation: it combines two designs to produce a design, rather than combining a design and the contents of the drawing plane to produce the new contents of the drawing plane. For purposes of color blending, the first argument to `compose-over` is called the foreground and the second argument is called the background. See Chapter 14 for the details of `compose-over`.

Color blending is defined by an ideal function $\mathcal{F}\colon (r_1, g_1, b_1, o_1, r_2, g_2, b_2, o_2) \rightarrow (r_3, g_3, b_3, o_3)$ that operates on the color and opacity at a single point. (r_1, g_1, b_1, o_1) are the foreground color and opacity. (r_2, g_2, b_2, o_2) are the background color and opacity. (r_3, g_3, b_3, o_3) are the resulting color and opacity. The color blending function $\mathcal{F}$ is conceptually applied at every point in the drawing plane.

$\mathcal{F}$ performs linear interpolation on all four components:

$$\begin{array}{rcrcl} o_3 & = & o_1 & + & (1 - o_1) * o_2 \\ r_3 & = & (o_1 * r_1 & + & (1 - o_1) * o_2 * r_2)/o_3 \\ g_3 & = & (o_1 * g_1 & + & (1 - o_1) * o_2 * g_2)/o_3 \\ b_3 & = & (o_1 * b_1 & + & (1 - o_1) * o_2 * b_2)/o_3 \end{array}$$

Note that if o_3 is zero, these equations would divide zero by zero. In that case r_3, g_3, and b_3 are defined to be zero.

CLIM requires that $\mathcal{F}$ be implemented exactly if o_1 is zero or one or if o_2 is zero. If o_1 is zero, the result is the background. If o_1 is one or o_2 is zero, the result is the foreground. For fractional opacity values, an implementation can deviate from the ideal color blending function either because the implementation has limited opacity resolution or because the implementation can compute a different color blending function much more quickly.

If a medium's background design is not completely opaque at all points, the consequences are unspecified. Consequently, a drawing plane is always opaque and drawing can use simplified color blending that assumes $o_2 = 1$ and $o_3 = 1$. However, `compose-over` must handle a non-opaque background correctly.

Note that these (r, g, b, o) quadruples of real numbers between 0 and 1 are mathematical and an implementation need not store information in this form. Most implementations are expected to use a different representation.

13.6 Indirect Inks

Drawing with an *indirect ink* is the same as drawing another design named directly. For example, `+foreground-ink+` is a design that draws the medium's foreground design and is the default value of the `:ink` drawing option. Indirect inks exist for the benefit of output recording. For example, one can draw with `+foreground-ink+`, change to a different `medium-foreground`, and replay the output record; the replayed output will come out in the new color.

You can change the foreground or background design of a medium at any time. This changes the contents of the medium's drawing plane. The effect is as if everything on the drawing plane is erased, the background design is drawn onto the drawing plane, and then everything that was ever drawn (provided it was saved in the output history) is drawn over again, using the medium's new foreground and background.

If an infinite recursion is created using an indirect ink, an error is signalled when the recursion is created, when the design is used for drawing, or both.

Two indirect inks are defined, but no advertised way is provided to create more of them.

⇒ `+foreground-ink+` [*Constant*]

An indirect ink that uses the medium's foreground design.

⇒ `+background-ink+` [*Constant*]

An indirect ink that uses the medium's background design.

13.7 Flipping Ink

⇒ `make-flipping-ink` *design1 design2* [*Function*]

Returns a design that interchanges occurrences of the two designs *design1* and *design2*. Drawing the resulting design over a background (either by drawing or with `compose-over`) is defined to change the color in the background that would have been drawn by *design1* at that point into the color that would have been drawn by *design2* at that point, and vice versa. The effect on any color other than the colors determined by those two designs is unspecified; however, drawing the same figure twice using the same flipping ink is guaranteed to be an "identity" operation. If either *design1* or *design2* is not solid, the consequences are unspecified. The purpose of flipping is to allow the use of "XOR hacks" for temporary changes to the display.

The opacity of a flipping ink is zero at points where the opacity of either *design1* or *design2* is zero. Otherwise the opacity of a flipping ink is 1. If *design1* or *design2* is translucent, the consequences are unspecified. If `compose-in` or `compose-out` is used to make a flipping ink translucent, the consequences are unspecified.

If *design1* and *design2* are equivalent, the result can be `+nowhere+`.

In Release 2, `make-flipping-ink` might require *design1* and *design2* to be colors.

⇒ `+flipping-ink+` [*Constant*]

A flipping ink that flips `+foreground-ink+` and `+background-ink+`.

13.8 Examples of Simple Drawing Effects

Drawing in the foreground color. Use the default, or specify `:ink +foreground-ink+`.

Erasing. Specify `:ink +background-ink+`.

Drawing in color. Specify `:ink +green+`, `:ink (make-rgb-color 0.6 0.0 0.4)`, and so forth.

Drawing an opaque gray. Specify `:ink (make-gray-color 0.25)` to draw in a shade of gray independent of the window's foreground color. On a non-color, non-grayscale display this will generally turn into a stipple.

Chapter 14

General Designs

This chapter discusses more general designs than Chapter 13 and reveals that regions are also designs. This chapter generalizes to those designs that do not have the same color and opacity at every point in the drawing plane. These include:

- composite designs,
- patterns,
- stencils,
- tiled designs,
- transformed designs,
- output record designs, and
- regions

Several of the features described in this chapter may not be fully supported in Release 2 of CLIM.

Note: A design is a unification of both a shape and a color and opacity. As such, a design can serve multiple roles. For example, the same design can play the role of an ink that colors the drawing plane, a shape that specifies where to draw another design, a stencil that controls the compositing of two designs, the background of a window, or a region that defines clipping. It is important not to get confused between *type*, which is inherent in an object, and *role*, which is determined by how an object is used in a particular function call.

14.1 The Compositing Protocol

Compositing creates a design whose appearance at each point is a composite of the appearances of two other designs at that point. Three varieties of compositing are provided: *composing over*, *composing in*, and *composing out*.

The methods for `compose-over`, `compose-in`, and `compose-out` will typically specialize both of the design arguments.

In Release 2, compositing might only be supported for uniform designs.

⇒ `compose-over` *design1 design2* [*Generic Function*]

Composes a new design that is equivalent to the *design design1* on top of the *design design2*. Drawing the resulting design produces the same visual appearance as drawing *design2* and then drawing *design1*, but might be faster and might not allow the intermediate state to be visible on the screen.

If both arguments are regions, `compose-over` is the same as `region-union`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified. The result returned by `compose-over` might be freshly constructed or might be an existing object.

⇒ `compose-in` *ink mask* [*Generic Function*]

Composes a new design by clipping the *design ink* to the inside of the *design mask*. The first design, *ink*, supplies the color, while the second design, *mask*, changes the shape of the design by adjusting the opacity.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the same color that *ink* specifies. The opacity is the opacity that *ink* specifies, multiplied by the stencil opacity of *mask*.

The *stencil opacity* of a design at a point is defined as the opacity that would result from drawing the design onto a fictitious medium whose drawing plane is initially completely transparent black (opacity and all color components are zero), and whose foreground and background are both opaque black. With this definition, the stencil opacity of a member of class `opacity` is simply its value.

If *mask* is a solid design, the effect of `compose-in` is to clip *ink* to *mask*. If *mask* is translucent, the effect is a soft matte.

If both arguments are regions, `compose-in` is the same as `region-intersection`.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified. The result returned by `compose-in` might be freshly constructed or might be an existing object.

⇒ `compose-out` *ink mask* [*Generic Function*]

Composes a new design by clipping the *design ink* to the outside of the *design mask*. The first design, *ink*, supplies the color, while the second design, *mask*, changes the shape of the design by adjusting the opacity.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the same color that *ink* specifies. The opacity is the opacity that *ink* specifies, multiplied by 1 minus the stencil opacity of *mask*.

If *mask* is a solid design, the effect of `compose-out` is to clip *ink* to the complement of *mask*. If *mask* is translucent, the effect is a soft matte.

If both arguments are regions, `compose-out` is the same as `region-difference` of *mask* and *ink*.

This function is permitted to capture its mutable inputs; the consequences of modifying those objects are unspecified. The result returned by `compose-out` might be freshly constructed or might be an existing object.

14.2 Patterns and Stencils

Patterning creates a bounded rectangular arrangement of designs, like a checkerboard. Drawing a pattern draws a different design in each rectangular cell of the pattern. To create an infinite pattern, apply `make-rectangular-tile` to a pattern.

A *stencil* is a special kind of pattern that contains only opacities.

⇒ `make-pattern` *array designs* [*Function*]

Returns a pattern design that has `(array-dimension` *array* `0)` cells in the vertical direction and `(array-dimension` *array* `1)` cells in the horizontal direction. *array* must be a two-dimensional array of non-negative integers less than the length of *designs*. *designs* must be a sequence of designs. The design in cell (i, j) of the resulting pattern is the nth element of *designs*, if n is the value of `(aref` *array* `i j)`. For example, *array* can be a bit-array and *designs* can be a list of two designs, the design drawn for 0 and the one drawn for 1.

Each cell of a pattern can be regarded as a hole that allows the design in it to show through. Each cell might have a different design in it. The portion of the design that shows through a hole is the portion on the part of the drawing plane where the hole is located. In other words, incorporating a design into a pattern does not change its alignment to the drawing plane, and does not apply a coordinate transformation to the design. Drawing a pattern collects the pieces of designs that show through all the holes and draws the pieces where the holes lie on the drawing plane. The pattern is completely transparent outside the area defined by the array.

Each cell of a pattern occupies a 1 by 1 square. You can use `transform-region` to scale the pattern to a different cell size and shape, or to rotate the pattern so that the rectangular cells become diamond-shaped. Applying a coordinate transformation to a pattern does not affect the designs that make up the pattern. It only changes the position, size, and shape of the cells' holes, allowing different portions of the designs in the cells to show through. Consequently, applying `make-rectangular-tile` to a pattern of nonuniform designs can produce a different appearance in each tile. The pattern cells' holes are tiled, but the designs in the cells are not tiled and a different portion of each of those designs shows through in each tile.

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

⇒ `pattern-width` *pattern* [*Generic Function*]
⇒ `pattern-height` *pattern* [*Generic Function*]

These functions return the width and height, respectively, of the pattern *pattern*.

⇒ `make-stencil` *array* [*Function*]

Returns a pattern design that has `(array-dimension` *array* `0)` cells in the vertical direction and `(array-dimension` *array* `1)` cells in the horizontal direction. *array* must be a two-dimensional array of real numbers between 0 and 1 (inclusive) that represent opacities. The design in cell (i, j) of the resulting pattern is the value of `(make-opacity (aref` *array* `i j))`.

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

14.3 Tiling

Tiling repeats a rectangular portion of a design throughout the drawing plane. This is most commonly used with patterns.

⇒ `make-rectangular-tile` *design width height* [*Function*]

Returns a design that, when used as an ink, tiles a rectangular portion of the design *design* across the entire drawing plane. The resulting design repeats with a period of *width* horizontally and *height* vertically. *width* and *height* must both be integers. The portion of *design* that appears in each tile is a rectangle whose top-left corner is at $(0, 0)$ and whose bottom-right corner is at $(width, height)$. The repetition of *design* is accomplished by applying a coordinate transformation to shift *design* into position for each tile, and then extracting a *width* by *height* portion of that design.

Applying a coordinate transformation to a rectangular tile does not change the portion of the argument *design* that appears in each tile. However, it can change the period, phase, and orientation of the repeated pattern of tiles. This is so that adjacent figures drawn using the same tile have their inks "line up".

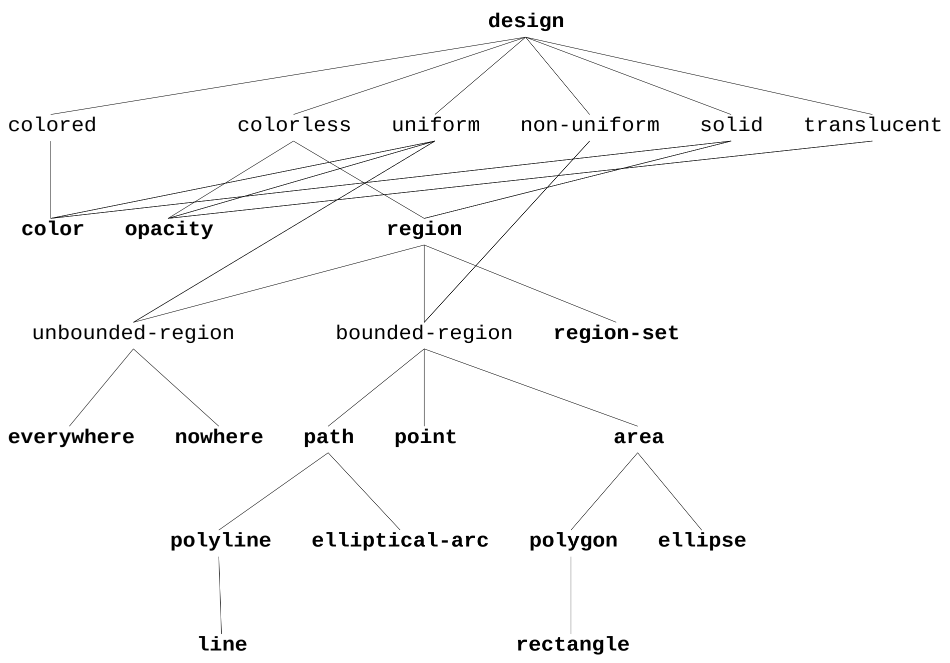


Figure 14.1: The class structure for all designs and regions. Entries in bold correspond to real CLIM classes.

14.4 Regions as Designs

Any member of the class `region` is a solid, colorless design. The design is opaque at points in the region and transparent elsewhere. Figure 14.1 shows how the design and region classes relate to each other.

Since bounded designs obey the region protocol, the functions `transform-region` and `untransform-region` accept any design as their second argument and apply a coordinate transformation to the design. The result is a design that might be freshly constructed or might be an existing object.

Transforming a uniform design simply returns the argument. Transforming a composite, flipping, or indirect design applies the transformation to the component design(s). Transforming a pattern, tile, or output record design is described in the sections on those designs.

14.5 Arbitrary Designs

⇒ `draw-design` *medium design* `&key` *ink clipping-region transformation line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape text-style text-family text-face text-size* [*Generic Function*]

Draws the *design design* onto the *medium medium.* This is defined to work for all types of regions and designs, although in practice some implementations may be more restrictive. *ink*, *transformation*, and *clipping-region* are used to modify the medium. The other drawing arguments control the drawing of the design, depending on what sort of design is being drawn. For instance, if *design* is a path, then line style options may be supplied.

If *design* is an area, `draw-design` paints the specified region of the drawing plane with medium's current ink. If *design* is a path, `draw-design` strokes the path with medium's current ink under control of the line-style. If *design* is a point, `draw-design` is the same as `draw-point`.

If *design* is a color or an opacity, `draw-design` paints the entire drawing plane (subject to the clipping region of the medium).

If *design* is `+nowhere+`, `draw-design` has no effect.

If *design* is a non-uniform design (see Chapter 14), `draw-design` paints the design, positioned at coordinates $(0, 0)$.

CLIM implementations are required to support `draw-design` for the following cases:

- Designs created by the geometric object constructors, such as `make-line` and `make-ellipse`, in conjunction with drawing arguments that supply the drawing ink.
- Designs created by calling `compose-in` on a color and an object created by a geometric object constructor.
- Designs created by calling `compose-over` on any of the cases above.
- Designs returned by `make-design-from-output-record`.

⇒ `draw-pattern*` *medium pattern x y* `&key` *clipping-region transformation* [*Function*]

Draws the pattern *pattern* on the medium *medium* at the position (x, y). *pattern* is any design created by `make-pattern`. *clipping-region* and *transformation* are as for `with-drawing-options` or any of the drawing functions.

Note that *transformation* only affects the position at which the pattern is drawn, not the pattern itself. If a programmer wishes to affect the pattern, he should explicity call `transform-region` on the pattern.

Drawing a bitmap consists of drawing an appropriately aligned and scaled pattern constructed from the bitmap's bits. A 1 in the bitmap corresponds to `+foreground-ink+`, while a 0 corresponds to `+background-ink+` if an opaque drawing operation is desired, or to `+nowhere+` if a transparent drawing operation is desired.

Drawing a (colored) raster image consists of drawing an appropriately aligned and scaled pattern constructed from the raster array and raster color map.

`draw-pattern*` could be implemented as follows, assuming that the functions `pattern-width` and `pattern-height` return the width and height of the pattern.

```
(defun draw-pattern* (medium pattern x y &key clipping-region transformation)
  (check-type pattern pattern)
  (let ((width (pattern-width pattern))
        (height (pattern-height pattern)))
    (if (or clipping-region transformation)
        (with-drawing-options (medium :clipping-region clipping-region
                                      :transformation transformation)
          (draw-rectangle* medium x y (+ x width) (+ y height)
                           :filled t :ink pattern))
        (draw-rectangle* medium x y (+ x width) (+ y height)
                         :filled t :ink pattern))))
```

14.6 Examples of More Complex Drawing Effects

Painting a gray or colored wash over a display. Specify a translucent design as the ink, such as `:ink (compose-in +black+ (make-opacity 0.25))`, `:ink (compose-in +red+ (make-opacity 0.1))`, or `:ink (compose-in +foreground-ink+ (make-opacity 0.75))`. The last example can be abbreviated as `:ink (make-opacity 0.75)`. On a non-color, non-grayscale display this will usually turn into a stipple.

Drawing a faded but opaque version of the foreground color. Specify `:ink (compose-over (compose-in +foreground-ink+ (make-opacity 0.25)) +background-ink+)` to draw at 25% of the normal contrast. On a non-color, non-grayscale display this will probably turn into a stipple.

Drawing a tiled pattern. Specify `:ink (make-rectangular-tile (make-pattern` *array colors*`))`.

Drawing a "bitmap". Use `(draw-design` *medium* `(make-pattern` *bit-array* `(list +background-ink+ +foreground-ink+)) :transformation (make-translation-transformation` *x y*`))`.

14.7 Design Protocol

Minor issue: *The generic functions underlying the functions described in this and the preceding chapter will be documented later. This will allow for programmer-defined design classes. This also needs to describe how to decode designs into inks. — SWM*

Part V

Extended Stream Output Facilities

Chapter 15

Extended Stream Output

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

15.1 Basic Output Streams

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

⇒ `standard-output-stream` [*Class*]

This class provides an implementation of the CLIM basic output stream protocol, based on the CLIM output kernel. Members of this class are mutable.

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

Writes the character *character* to the *output stream stream*, and returns *character* as its value.

⇒ `stream-write-string` *stream string* `&optional` *(start* `0`*) end* [*Generic Function*]

Writes the string *string* to the *output stream stream*. If *start* and *end* are supplied, they are integers that specify what part of *string* to output. *string* is returned as the value.

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

Writes an end of line character on the *output stream stream*, and returns *false*.

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

Writes an end of line character on the *output stream stream* only if the stream is not at the beginning of the line.

⇒ `stream-finish-output` *stream* [*Generic Function*]

Ensures that all the output sent to the *output stream stream* has reached its destination, and only then return *false*.

⇒ `stream-force-output` *stream* [*Generic Function*]

Like `stream-finish-output`, except that it may immediately return *false* without waiting for the output to complete.

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

Aborts any outstanding output operation in progress on the *output stream stream*, and returns *false*.

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

This function returns the column number where the next character will be written on the *output stream stream*. The first column on a line is numbered 0.

⇒ `stream-start-line-p` *stream* [*Generic Function*]

Returns *true* if the *output stream stream* is positioned at the beginning of a line (that is, column 0), otherwise returns *false*.

⇒ `stream-advance-to-column` *stream column* [*Generic Function*]

Writes enough blank space on the *output stream stream* so that the next character will be written at the position specified by *column*, which is an integer.

15.2 Extended Output Streams

In addition to the basic output stream protocol, CLIM defines an extended output stream protocol. This protocol extends the stream model to maintain the state of a text cursor, margins, text styles, inter-line spacing, and so forth.

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

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

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

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

⇒ `:foreground` [*Initarg*]
⇒ `:background` [*Initarg*]
⇒ `:text-style` [*Initarg*]
⇒ `:vertical-spacing` [*Initarg*]
⇒ `:text-margin` [*Initarg*]
⇒ `:end-of-line-action` [*Initarg*]
⇒ `:end-of-page-action` [*Initarg*]
⇒ `:default-view` [*Initarg*]

All subclasses of `extended-output-stream` must handle these initargs, which are used to specify, respectively, the medium foreground and background, default text style, vertical spacing, default text margin, end of line and end of page actions, and the default view for the stream.

`:foreground`, `:background`, and `:text-style` are handled via the usual pane initialize options (see Section 29.2.1).

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

This class provides an implementation of the CLIM extended output stream protocol, based on the CLIM output kernel.

Members of this class are mutable.

15.3 The Text Cursor

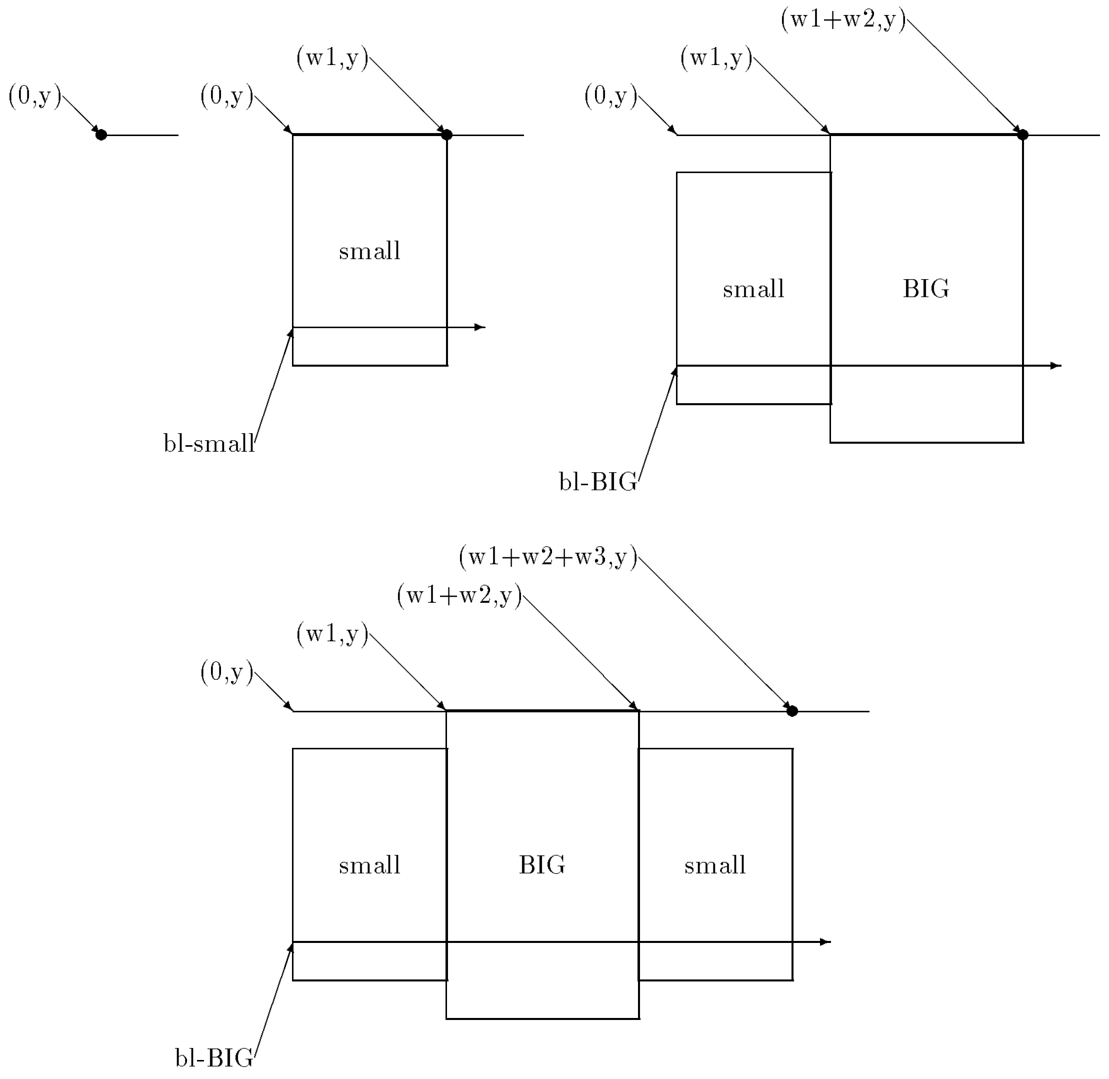
In the days when display devices displayed only two dimensional arrays of fixed width characters, the text cursor was a simple thing. A discrete position was selected in integer character units, and a character could go there and noplace else. Even for variable width fonts, simply addressing a character by the pixel position of one of its corners is sufficient. However, variable height fonts with variable baselines on pixel-addressable displays upset this simple model. The "logical" vertical reference point is the baseline, as it is in typesetting. In typesetting, however, an entire line of text is created with baselines aligned and padded to the maximum ascent and descent, then the entire line is put below the previous line.

It is clearly desirable to have the characters on a line aligned with their baselines, but when the line on the display is formed piece by piece, it is impossible to pick in advance the proper baseline. The solution CLIM adopts is to choose a baseline, but not commit to it.

The CLIM model says that text has at least 6 properties. With a reference point of $(0, 0)$ at the upper left of the text, it has a bounding box consisting of ascent, descent, left kerning, right extension, and a displacement to the next reference point in both x and y. CLIM determines the position of the reference point and draws the text relative to that, and then the cursor position is adjusted by the displacement. In this way, text has width and height, but the x and y displacements need not equal the width and height.

CLIM adopts the following approach to the actual rendering of a glyph. Textual output using the stream functions (*not* the graphics functions) maintains text on a "line". Note that a line is not an output record, but is rather a collection of “text so far”, a top (which is positioned at the bottom of the previous line plus the stream’s vertical spacing), a baseline, a bottom, and a “cursor position”. The cursor position is defined to be at the top of the line, not at the baseline. The reason for this is that the baseline can move, but the top is relative to the previous line, which has been completed and therefore doesn’t move. If text is drawn on the current line whose ascent is greater than the current ascent of the line, then the line is moved down to make room. This can be done easily using the output records for the existing text on the line. When there is enough room, the reference point for the text is the x position of the cursor at the baseline, and the cursor position is adjusted by the displacement.

The following figures show this in action before and after each of three characters are drawn. In all three figure, the small circle is the “cursor position”. At first, there is nothing on the line. The first character establishes the initial baseline, and is then drawn. The upper left corner of the character is where the cursor was (as in the traditional model), but this will not remain the case. Drawing the second character, which is larger than the first, requires moving the first character down in order to get the baselines to align; during this time, the top of the line remains the same. Again, the upper left of the second character is where the cursor was, but that is no longer the case for the first character (which has moved down). The third character is smaller than the second, so no moving of characters needs to be done. However, the character is drawn to align the baselines, which in this case means the upper left is *not* where the cursor was. Nor is the cursor at the upper right of the character as it was for the previous two characters. It is, however, at the upper right of the collective line.



Minor issue: *The above may be too simplistic. The displacement probably wants to depend not only on language but language rendering mode. For example, Japanese can apparently go either vertically or horizontally. It may be necessary to have the bounding box and perhaps the reference point dispatch as well. Similarly, "newline" could mean "down one line, all the way to the left" for English, "down one line, all the way to the right" for Arabic, or "to the left one line, all the way to the top." "Home cursor" is another ditty to consider. We need to discuss this on a larger scale with people who know multi-lingual rendering issues. — DCPL*

15.3.1 Text Cursor Protocol

Many streams that maintain a text cursor display some visible indication of the text cursor. The object that represents this display is (somewhat confusingly) also called a cursor.

An *active* cursor is one that is being actively maintained by its owning sheet. An active cursor has a *state* that is either on or off. An active cursor also has a piece of state that indicates the the owning sheet has the input focus.

⇒ `cursor` [*Protocol Class*]

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

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

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

⇒ `:sheet` [*Initarg*]

The `:sheet` initarg is used to specify the sheet with which the cursor is associated.

⇒ `standard-text-cursor` [*Class*]

The instantiable class that implements a text cursor. Typically, ports will further specialize this class.

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

Returns the sheet with which the *cursor cursor* is associated.

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

Returns the x and y position of the *cursor cursor* as two values. x and y are in the coordinate system of the cursor's sheet.

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

Sets the x and y position of the *cursor cursor* to the specified position. x and y are in the coordinate system of the cursor's sheet.

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

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

Returns (or sets) the "active" attribute of the cursor. When *true*, the cursor is active.

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

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

Returns (or sets) the "state" attribute of the cursor. When *true*, the cursor is visible. When *false*, the cursor is not visible.

⇒ `cursor-focus` *cursor* [*Generic Function*]

Returns the "focus" attribute of the cursor. When *true*, the sheet owning the cursor has the input focus.

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

These are convenience functions that combine the functionality of `cursor-active` and `cursor-state`. The visibility can be either `:on` (meaning that the cursor is active and visible at its current position), `:off` (meaning that the cursor is active, but not visible at its current position), or `nil` (meaning that the cursor is not activate).

15.3.2 Stream Text Cursor Protocol

The following generic functions comprise the stream text cursor protocol. Any extended output stream class must implement methods for these generic functions.

⇒ `stream-text-cursor` *stream* [*Generic Function*]

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

Returns (or sets) the text cursor object for the stream *stream*.

⇒ `stream-cursor-position` *stream* [*Generic Function*]

Returns the current text cursor position for the *extended output stream stream* as two integer values, the x and y positions.

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

Sets the text cursor position of the *extended output stream stream* to x and y. x and y are in device units, and must be integers.

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

⇒ `stream-increment-cursor-position` *stream dx dy* [*Generic Function*]

Moves the text cursor position of the *extended output stream stream* relatively, adding dx to the x coordinate and dy to the y coordinate. Either of dx or dy may be `nil`, meaning the the x or y cursor position will be unaffected. Otherwise, dx and dy must be integers.

15.4 Text Protocol

The following generic functions comprise the text protocol. Any extended output stream class must implement methods for these generic functions.

⇒ `stream-character-width` *stream character* `&key` *text-style* [*Generic Function*]

Returns a rational number corresponding to the amount of horizontal motion of the cursor position that would occur if the character *character* were output to the *extended output stream stream* in the *text style text-style* (which defaults to the current text style for the stream). This ignores the stream's text margin.

⇒ `stream-string-width` *stream character* `&key` *start end text-style* [*Generic Function*]

Computes how the cursor position would move horizontally if the string *string* were output to the *extended output stream stream* in the *text style text-style* (which defaults to the current text style for the stream) starting at the left margin. This ignores the stream's text margin.

The first returned value is the x coordinate that the cursor position would move to. The second returned value is the maximum x coordinate the cursor would visit during the output. (This is the same as the first value unless the string contains a `#\Newline` character.)

start and *end* are integers, and default to 0 and the length of the string, respectively.

⇒ `stream-text-margin` *stream* [*Generic Function*]
⇒ `(setf stream-text-margin)` *margin stream* [*Generic Function*]

The x coordinate at which text wraps around on the *extended output stream stream* (see `stream-end-of-line-action`). The default setting is the width of the viewport, which is the right-hand edge of the viewport when it is horizontally scrolled to the "initial position".

You can use `setf` with `stream-text-margin` to establish a new text margin. If *margin* is `nil`, then the width of the viewport will be used. If the width of the viewport is later changed, the text margin will change, too.

⇒ `stream-line-height` *stream* `&key` *text-style* [*Generic Function*]

Returns what the line height of a line on the *extended output stream stream* containing text in the *text style text-style* would be, as a rational number. The height of the line is measured from the baseline of the text style to its ascent. *text-style* defaults to the current text style for the stream.

⇒ `stream-vertical-spacing` *stream* [*Generic Function*]

Returns the current inter-line spacing (as a rational number) for the *extended output stream stream.*

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

Returns the current text baseline (as a rational number) for the *extended output stream stream.*

15.4.1 Mixing Text and Graphics

The following macro provides a convenient way to mix text and graphics on the same output stream.

⇒ `with-room-for-graphics` *(*`&optional` *stream* `&key` *(first-quadrant* `t`*) height (move-cursor* `t`*) record-type)* `&body` *body* [*Macro*]

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

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

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

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

15.4.2 Wrapping of Text Lines

⇒ `stream-end-of-line-action` *stream* [*Generic Function*]
⇒ `(setf stream-end-of-line-action)` *action stream* [*Generic Function*]

The end-of-line action controls what happens if the text cursor position moves horizontally out of the viewport, or if text output reaches the text margin. (By default the text margin is the width of the viewport, so these are usually the same thing.)

`stream-end-of-line-action` returns the end-of-line action for the *extended output stream stream*. It can be changed by using `setf` on `stream-end-of-line-action`.

The end-of-line action is one of:

- `:wrap`—when doing text output, wrap the text around (that is, break the text line and start another line). When setting the cursor position, scroll the window horizontally to keep the cursor position inside the viewport. This is the default.
- `:scroll`—scroll the window horizontally to keep the cursor position inside the viewport, then keep doing the output.

- `:allow`—ignore the text margin and do the output on the drawing plane beyond the visible part of the viewport.

⇒ `with-end-of-line-action` *(stream action)* `&body` *body* [*Macro*]

Temporarily changes *stream*'s end-of-line action for the duration of execution of body. *action* must be one of the actions described in `stream-end-of-line-action`.

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

⇒ `stream-end-of-page-action` *stream* [*Generic Function*]
⇒ `(setf stream-end-of-page-action)` *action stream* [*Generic Function*]

The end-of-page action controls what happens if the text cursor position moves vertically out of the viewport.

`stream-end-of-page-action` returns the end-of-page action for the *extended output stream stream*. It can be changed by using `setf` on `stream-end-of-page-action`.

The end-of-page action is one of:

- `:scroll`—scroll the window vertically to keep the cursor position inside the viewport, then keep doing output. This is the default.
- `:allow`—ignore the viewport and do the output on the drawing plane beyond the visible part of the viewport.
- `:wrap`—when doing text output, wrap the text around (that is, go back to the top of the viewport).

⇒ `with-end-of-page-action` *(stream action)* `&body` *body* [*Macro*]

Temporarily changes *stream*'s end-of-page action for the duration of execution of body. *action* must be one of the actions described in `stream-end-of-page-action`.

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

15.5 Attracting the User's Attention

⇒ `beep &optional` *medium* [*Generic Function*]

Attracts the user's attention, usually with an audible sound.

15.6 Buffering of Output

Some mediums that support the output protocol may buffer output. When buffering is enabled on a medium, the time at which output is actually done on the medium is unpredictable. `force-output` or `finish-output` can be used to ensure that all pending output gets completed. If the medium is a bidirectional stream, a `force-output` is performed whenever any sort of input is requested on the stream.

`with-buffered-output` provides a way to control when buffering is enabled on a medium. By default, CLIM's interactive streams are buffered if the underlying window system supports buffering.

⇒ `medium-buffering-output-p` *medium* [*Generic Function*]

Returns *true* if the *medium medium* is currently buffering output, otherwise returns *false*.

⇒ `(setf medium-buffering-output-p)` *buffer-p medium* [*Generic Function*]

Sets `medium-buffering-output-p` of the *medium medium* to *buffer-p*.

⇒ `with-output-buffered` *(medium* `&optional` *(buffer-p* `t`*))* `&body` *body* [*Macro*]

If *buffer-p* is *true* (the default), this causes the *medium* designated by *medium* to start buffering output, and evaluates *body* in that context. If *buffer-p* is *false*, `force-output` will be called before *body* is evaluated. When *body* is exited (or aborted from), `force-output` will be called if output buffering will be disabled after `with-output-buffered` is exited.

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

Chapter 16

Output Recording

16.1 Overview of Output Recording

CLIM provides a mechanism whereby output (textual and graphical) may be captured into an *output history* for later replay on the same stream. This mechanism serves as the basis for many other tools, such as the formatted output and presentation mechanisms described elsewhere.

The output recording facility is layered on top of the graphics and text output protocols. It works by intercepting the operations in the graphics and text output protocols, and saving information about these operations in objects called *output records*. In general, an output record is a kind of display list, that is, a collection of instructions for drawing something on a stream. Some output records may have *children*, that is, a collection of inferior output records. Other output records, which are called *displayed output records*, correspond directly to displayed information on the stream, and do not have children. If you think of output records being arranged in a tree, displayed output records are all of the leaf nodes in the tree, for example, displayed text and graphics records.

Displayed output records must record the state of the supplied drawing options at the instant the output record is created, as follows. The ink supplied by the programmer must be captured without resolving indirect inks; this is so that a user can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay. The effect of the specified "user" transformation (composed with the medium transformation) must be captured; CLIM implementations are free to do this either by saving the transformation object or by saving the transformed values of all objects that are affected by the transformation. The user clipping region and line style must be captured in the output record as well. Subsequent replaying of the record under a new user transformation, clipping region, or line style will not affect the replayed output. CLIM implementation are permitted to capture the text style either fully merged against the medium's default, or not; in the former case, subsequent changes to the medium's default text style will not affect replaying the record, but in the latter case changing the default text style will affect replaying.

A CLIM stream that supports output recording has an output history object, which is a special kind of output record that supports some other operations. CLIM defines a standard set of output history implementations and a standard set of output record types.

The output recording mechanism is defined so as to permit application specific or host window system specific implementations of the various recording protocols. CLIM implementations should provide several types of standard output records with different characteristics for search, storage, and retrieval. Two examples are "sequence" output records (which store elements in a sequence, and whose insertion and retrieval complexity is O(n)) and "tree" output records (which store elements in some sort of tree based on the location of the element, and whose insertion and retrieval complexity is O(log n)).

Major issue: *There is a proposal on the table to unify the sheet and output record protocols, not by unifying the class structure, but by making them implement the same generic functions where that makes sense. For instance, sheets and output records both have regions, transformations (that relate sheets to their parents), both support a repainting operation, and so forth.*

In particular, `sheet-parent` *and* `output-record-parent` *are equivalent, as are* `sheet-children` *and* `output-record-children`, `sheet-adopt-child` *and* `add-output-record`, `sheet-disown-child` *and* `delete-output-record`, *and* `repaint-sheet` *and* `replay-output-record`, *and the mapping functions.* `output-record-position` *and its* `setf` *function have sheet analogs. The sheet and output record notification functions are also equivalent.*

This simplifies the conceptual framework of CLIM, and could eventually simplify the implementation as well. Doing this work now opens the door for later unifications, such unifying the pane layout functionality with table formatting. — York, SWM

16.2 Output Records

⇒ `output-record` [*Protocol Class*]

The protocol class that is used to indicate that an object is an output record, that is, an object that contains other output records. This is a subclass of `bounding-rectangle`, and as such, output records obey the bounding rectangle protocol. If you want to create a new class that behaves like an output record, it should be a subclass of `output-record`. All instantiable subclasses of `output-record` must obey the output record protocol.

All output records are mutable.

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

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

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

The protocol class that is used to indicate that an object is a displayed output record, that is, an object that represents a visible piece of output on some output stream. This is a subclass of `bounding-rectangle`. If you want to create a new class that behaves like a displayed output record, it should be a subclass of `displayed-output-record`. All instantiable subclasses of `displayed-output-record` must obey the displayed output record protocol.

All displayed output records are mutable.

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

Returns *true* if *object* is a *displayed output record*, otherwise returns *false*.

⇒ `:x-position` [*Initarg*]
⇒ `:y-position` [*Initarg*]
⇒ `:parent` [*Initarg*]

All subclasses of either `output-record` or `displayed-output-record` must handle these three initargs, which are used to specify, respectively, the x and y position of the output record, and the parent of the output record.

⇒ `:size` [*Initarg*]

All subclasses of `output-record` must handle the `:size` initarg. It is used to specify how much room should be left for child output records (if, for example, the children are stored in a vector). It is permissible for `:size` to be ignored, provided that the resulting output record is able to store the specified number of child output records.

16.2.1 The Basic Output Record Protocol

All subclasses of `output-record` and `displayed-output-record` must inherit or implement methods for the following generic functions.

When the generic functions in this section take both *record* and a *stream* arguments, CLIM implementations will specialize the *stream* argument for the `standard-output-recording-stream` class and the *record* argument for all of the implementation-specific output record classes.

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

Returns the x and y position of the *output record record* as two rational numbers. The position of an output record is the position of the upper-left corner of its bounding rectangle. The position is relative to the stream, where $(0, 0)$ is (initially) the upper-left corner of the stream.

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

Changes the x and y position of the *output record record* to be x and y (which are rational numbers), and updates the bounding rectangle to reflect the new position (and saved cursor positions, if the output record stores it). If *record* has any children, all of the children (and their descendants as well) will be moved by the same amount as *record* was moved. The bounding rectangles of all of *record*'s ancestors will also be updated to be large enough to contain *record*. This does not replay the output record, but the next time the output record is replayed it will appear at the new position.

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

⇒ `output-record-start-cursor-position` *record* [*Generic Function*]

Returns the x and y starting cursor position of the *output record record* as two integer values. The positions are relative to the stream, where $(0, 0)$ is (initially) the upper-left corner of the stream.

Text output records and updating output records maintain the cursor position. Graphical output records and other output records that do not require or affect the cursor position will return `nil` as both of the values.

⇒ `(setf* output-record-start-cursor-position)` *x y record* [*Generic Function*]

Changes the x and y starting cursor position of the *output record record* to be x and y (which are integers). This does not affect the bounding rectangle of *record*, nor does it replay the output record. For those output records that do not require or affect the cursor position, the method for this function is a no-op.

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

⇒ `output-record-end-cursor-position` *record* [*Generic Function*]

Returns the x and y ending cursor position of the *output record record* as two integer values. The positions are relative to the stream, where $(0, 0)$ is (initially) the upper-left corner of the stream. Graphical output records do not track the cursor position, so only text output record (and some others) will return meaningful values for this.

Text output records and updating output records maintain the cursor position. Graphical output records and other output records that do not require or affect the cursor position will return `nil` as both of the values.

⇒ `(setf* output-record-end-cursor-position)` *x y record* [*Generic Function*]

Changes the x and y ending cursor position of the *output record record* to be x and y (which are integers). This does not affect the bounding rectangle of *record*, nor does it replay the output record. For those output records that do not require or affect the cursor position, the method for this function is a no-op.

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

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

Returns the output record that is the parent of the *output record record*, or `nil` if the record has no parent.

⇒ `replay` *record stream* `&optional` *region* [*Function*]

This function bind `stream-recording-p` of *stream* to *false*, and then calls `replay-output-record` on the arguments *record*, *stream*, and *region*. If `stream-drawing-p` of *stream* is *false*, `replay` does nothing. `replay` is typically called during scrolling, by repaint handlers, and so on.

CLIM implementations are permitted to default *region* either to `nil` or to the region corresponding to viewport of *stream*.

⇒ `replay-output-record` *record stream* `&optional` *region x-offset y-offset* [*Generic Function*]

Displays the output captured by the *output record record* on the *output recording stream stream*, exactly as it was originally captured (subject to subsequent modifications). The current user transformation, line style, text style, ink, and clipping region of *stream* are all ignored during the replay operation. Instead, these are gotten from the output record.

If *record* is not a displayed output record, then replaying it involves replaying all of its children. If *record* is a displayed output record, then replaying it involves redoing the graphics operation captured in the record.

region is a region that can be supplied to limit what records are displayed. Only those records that overlap *region* are replayed. The default for *region* is `+everywhere+`.

It is permissible for implementations to restrict `replay-output-record` such that *stream* must be the same stream on which the output records were originally recorded.

Minor issue: *How does replaying a text output record (or any record that maintains the cursor position) affect the cursor position of the stream? It is probably that case that* `replay` *should not affect the cursor position. — SWM*

⇒ `output-record-hit-detection-rectangle*` *record* [*Generic Function*]

This method is used to establish the usual "effective size" of *record* for hit detection queries. Four values are returned corresponding to the edges of the bounding rectangle that is the hit detection rectangle. The default method (on CLIM's standard output record class) is equivalent to calling calling `bounding-rectangle*` on *record*, but this method can be specialized to return a larger bounding rectangle in order to implement a form of hysteresis.

⇒ `output-record-refined-position-test` *record x y* [*Generic Function*]

This is used to definitively answer hit detection queries, that is, determining that the point (x, y) is contained within the output record *record*. Once the position (x, y) has been determined to lie within `output-record-hit-detection-rectangle*`, `output-record-refined-sensitivity-test` is invoked. Output record subclasses can provide a method that provides a more precise definition of a hit, for example, output records for elliptical rings will implement a method that detects whether the pointing device is on the elliptical ring.

⇒ `highlight-output-record` *record stream state* [*Generic Function*]

This method is called in order to draw a highlighting box around the *output record record* on the *output recording stream stream*. *state* will be either `:highlight` (meaning to draw the highlighting) or `:unhighlight` (meaning to erase the highlighting). The default method (on

CLIM's standard output record class) will simply draw a rectangle that corresponds the the bounding rectangle of *record*.

⇒ `displayed-output-record-ink` *displayed-output-record* [*Generic Function*]

Returns the ink used by the *displayed output record displayed-output-record*.

16.2.2 The Output Record "Database" Protocol

All classes that are subclasses of `output-record` must implement methods for the following generic functions.

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

Returns a sequence of all of the children of the *output record record*. It is unspecified whether the sequence is a freshly created object or a "live" object representing the state of *record*.

⇒ `add-output-record` *child record* [*Generic Function*]

Adds the new *output record child* to the *output record record*. The bounding rectangle for *record* (and all its ancestors) must be updated to account for the new child record.

The methods for the `add-output-record` will typically specialize only the *record* argument.

⇒ `delete-output-record` *child record* `&optional` *(errorp* `t`*)* [*Generic Function*]

Removes the *output record child* from the *output record record*. The bounding rectangle for *record* (and all its ancestors) may be updated to account for the child having been removed, although this is not mandatory.

If *errorp* is *true* (the default) and *child* is not contained within *record*, then an error is signalled.

The methods for the `delete-output-record` will typically specialize only the *record* argument.

CLIM implementations are permitted not to manage the interaction between `delete-output-record` and the output record mapping functions.

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

Removes all of the children from the *output record record*, and resets the bounding rectangle of *record* to its initial state.

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

Returns the number of children contained within the *output record record*.

⇒ `map-over-output-records-containing-position` *function record x y* `&optional` *x-offset y-offset* `&rest` *function-args* [*Generic Function*]

Maps over all of the children of the *output record record* that contain the point at (x, y), calling *function* on each one. *function* is a function of one or more arguments, the first argument being the record containing the point; it has dynamic extent. *function* is also called with all of *function-args* as "apply" arguments.

If there are multiple records that contain the point and that overlap each other, `map-over-output-records-containing-position` must hit the uppermost (most recently inserted) record first and the bottommost (least recently inserted) record last. Otherwise, the order in which the records are traversed is unspecified.

`map-over-output-records-containing-position` returns `nil`.

⇒ `map-over-output-records-overlapping-region` *function record region* `&optional` *x-offset y-offset* `&rest` *function-args* [*Generic Function*]

Maps over all of the children of the *output record record* that overlap the *region region*, calling *function* on each one. *function* is a function of one or more arguments, the first argument being the record overlapping the region; it has dynamic extent. *function* is also called with all of *function-args* as "apply" arguments.

If there are multiple records that overlap the region and that overlap each other, `map-over-output-records-overlapping-region` must hit the bottommost (least recently inserted) record first and the uppermost (most recently inserted) record last. Otherwise, the order in which the records are traversed is unspecified.

`map-over-output-records-overlapping-region` returns `nil`.

16.2.3 Output Record Change Notification Protocol

The following functions are called by programmers and by CLIM itself in order to notify a parent output record when the bounding rectangle of one of its child output record changes.

⇒ `recompute-extent-for-new-child` *record child* [*Generic Function*]

This function is called whenever a new child is added to an output record. Its contract is to update the bounding rectangle of the *output record record* to be large enough to completely contain the new child *output record child*. The parent of *record* must be notified by calling `recompute-extent-for-changed-child`.

`add-output-record` is required to call `recompute-extent-for-new-child`.

⇒ `recompute-extent-for-changed-child` *record child old-min-x old-min-y old-max-x old-max-y* [*Generic Function*]

This function is called whenever the bounding rectangle of one of the childs of a record has been changed. Its contract is to update the bounding rectangle of the *output record record* to be large enough to completely contain the new bounding rectangle of the child *output record child*.

All of the ancestors of *record* must be notified by recursively calling `recompute-extent-for-changed-child`.

Whenever the bounding rectangle of an output record is changed or `delete-output-record` is called, `recompute-extent-for-changed-child` must be called to inform the parent of the record that a change has taken place.

⇒ `tree-recompute-extent` *record* [*Generic Function*]

This function is called whenever the bounding rectangles of a number of children of a record have been changed, such as happens during table and graph formatting. Its contract is to compute the bounding rectangle large enough to contain all of the children of the output record *record*, adjust the bounding rectangle of the *output record record* accordingly, and then call `recompute-extent-for-changed-child` on *record*.

16.3 Types of Output Records

This section discusses several types of output records, including two standard classes of output records and the displayed output record protocol.

16.3.1 Standard Output Record Classes

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

The standard instantiable class provided by CLIM to store a relatively short sequence of output records; a subclass of `output-record`. The insertion and retrieval complexity of this class is O(n). Most of the formatted output facilities (such as `formatting-table`) create output records that are a subclass of `standard-sequence-output-record`.

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

The standard instantiable class provided by CLIM to store longer sequences of output records. Typically, the child records of a tree output record will be maintained in some sort of sorted order, such as a lexicographic ordering on the x and y coordinates of the children. The insertion and retrieval complexity of this class is O(log n).

16.3.2 Graphics Displayed Output Records

Graphics displayed output records are used to record the output produced by the graphics functions, such as `draw-line*`. Each graphics displayed output record describes the output produced by a call to one of the graphics functions.

The exact contents of graphics displayed output records is unspecified, but they must store sufficient information to be able to exactly redraw the original output at replay time. The minimum information that must be captured for all graphics displayed output records is as follows:

- The description of the graphical object itself, for example, the end points of a line or the center point and radius of a circle.
- The programmer-supplied ink at the time the drawing function was called. Indirect inks must not be resolved, so that a user can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay.
- For paths, the programmer-supplied line-style at the time the drawing function was called.
- The programmer-supplied clipping region at the time the drawing function was called.
- The user transformation. This may be accomplished by either explicitly storing the transformation, or by transforming the coordinates supplied to the drawing function and capturing the transformed coordinates.

⇒ `graphics-displayed-output-record` [*Protocol Class*]

The protocol class that corresponds to output records for the graphics functions, such as `draw-line*`. This is a subclass of `displayed-output-record`. If you want to create a new class that behaves like a graphics displayed output record, it should be a subclass of `graphics-displayed-output-record`. All instantiable subclasses of `graphics-displayed-output-record` must obey the graphics displayed output record protocol.

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

Returns *true* if *object* is a *graphics displayed output record*, otherwise returns *false*.

16.3.3 Text Displayed Output Record

Text displayed output records are used to record the textual output produced by such functions as `stream-write-char` and `stream-write-string`. Each text displayed output record corresponds to no more than one line of textual output (that is, line breaks caused by `terpri` and `fresh-line` create a new text output record, as do certain other stream operations described below).

The exact contents of text displayed output records is unspecified, but they must store sufficient information to be able to exactly redraw the original output at replay time. The minimum information that must be captured for all text displayed output records is as follows:

- The displayed text strings.
- The starting and ending cursor positions.
- The text style in which the text string was written. Depending on the CLIM implementation, this may be either fully merged against the medium's default or not; in the former

case, subsequent changes to the medium's default text style will not affect replaying the record, but in the latter case changing the default text style will affect replaying.

- The programmer-supplied ink at the time the drawing function was called. Indirect inks must not be resolved, so that a user can later change the default foreground and background ink of the medium and have that change affect the already-created output records during replay.
- The programmer-supplied clipping region at the time the drawing function was called.

⇒ `text-displayed-output-record` [*Protocol Class*]

The protocol class that corresponds to text displayed output records. This is a subclass of **`displayed-output-record`**. If you want to create a new class that behaves like a text displayed output record, it should be a subclass of **`text-displayed-output-record`**. All instantiable subclasses of **`text-displayed-output-record`** must obey the text displayed output record protocol.

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

Returns *true* if *object* is a *text displayed output record*, otherwise returns *false*.

The following two generic functions comprise the text displayed output record protocol.

⇒ `add-character-output-to-text-record` *text-record character text-style width height baseline* [*Generic Function*]

Adds the character *character* to the *text displayed output record text-record* in the text style *text-style*. *width* and *height* are the width and height of the character in device units, and are used to compute the bounding rectangle for the text record. *baseline* is the new baseline for characters in the output record.

⇒ `add-string-output-to-text-record` *text-record string start end text-style width height baseline* [*Generic Function*]

Adds the string *string* to the *text displayed output record text-record* in the text style *text-style*. *start* and *end* are integers that specify the substring within *string* to add to the text output record. *width* and *height* are the width and height of the character in device units, and are used to compute the bounding rectangle for the text record. *baseline* is the new baseline for characters in the output record.

⇒ `text-displayed-output-record-string` *text-record* [*Generic Function*]

Returns the string contained by the *text displayed output record text-record*. This function returns objects that reveal CLIM's internal state; do not modify those objects.

16.3.4 Top-Level Output Records

Top-level output records are similar to ordinary output records, except that they must maintain additional state, such as the information required to display scroll bars.

⇒ `stream-output-history-mixin` [*Class*]

This class is mixed into some other output record class to produce a new class that is suitable for use as a a top-level output history. This class is not intended to be instantiated.

When the bounding rectangle of an member of this class is updated, CLIM implementations must update any window decorations (such as scroll bars) associated with the stream with which the *output record history* is associated.

⇒ `standard-tree-output-history` [*Class*]

The standard instantiable class provided by CLIM to use as the top-level output history. This will typically be a subclass of both `standard-tree-output-record` and `stream-output-history-mixin`.

⇒ `standard-sequence-output-history` [*Class*]

Another instantiable class provided by CLIM to use for top-level output records that have only a small number of children. This will typically be a subclass of both `standard-sequence-output-record` and `stream-output-history-mixin`.

16.4 Output Recording Streams

CLIM defines an extension to the stream protocol that supports output recording. The stream has an associated output history record and provides controls to enable and disable output recording.

Minor issue: *Do we want to support only output recording streams, or do we want to support output recording sheets as well? If the latter, we need to split apart graphics output recording and textual output recording, and rename lots of things. — SWM*

⇒ `output-recording-stream` [*Protocol Class*]

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

⇒ `output-recording-stream-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is an *output recording stream*, otherwise returns *false*.

⇒ `standard-output-recording-stream` [*Class*]

The class used by CLIM to implement output record streams. This is a subclass of `output-recording-stream`. Members of this class are mutable.

16.4.1 The Output Recording Stream Protocol

The following generic functions comprise the output recording stream protocol. All subclasses of `output-recording-stream` must implement methods for these generic functions.

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

Returns *true* when the *output recording stream stream* is recording all output performed to it, otherwise returns *false*.

⇒ `(setf stream-recording-p)` *recording-p stream* [*Generic Function*]

Changes the state of `stream-recording-p` to be *recording-p*, which must be either `t` or `nil`.

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

Returns *true* when the *output recording stream stream* will actually draw on the viewport when output is being performed to it, otherwise returns *false*.

⇒ `(setf stream-drawing-p)` *drawing-p stream* [*Generic Function*]

Changes the state of `stream-recording-p` to be *drawing-p*, which must be either `t` or `nil`.

⇒ `stream-output-history` *stream* [*Generic Function*]

Returns the history (or top-level output record) for the *output recording stream stream*.

⇒ `stream-current-output-record` *stream* [*Generic Function*]

The current "open" output record for the *output recording stream stream*, the one to which `stream-add-output-record` will add a new child record. Initially, this is the same as `stream-output-history`. As nested output records are created, this acts as a "stack".

⇒ `(setf stream-current-output-record)` *record stream* [*Generic Function*]

Sets the current "open" output record for the *output recording stream stream* to the *output record record*.

⇒ `stream-add-output-record` *stream record* [*Generic Function*]

Adds the *output record record* to the current output record on the *output recording stream stream* (that is, `stream-current-output-record`).

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

Directs the *output recording stream stream* to invoke `replay` on its output history. Only those records that overlap the *region region* (which defaults to the viewport of the stream) are replayed.

⇒ `erase-output-record` *record stream* `&optional` *(errorp* `t`*)* [*Generic Function*]

Erases the *output record record* from the *output recording stream stream*, removes *record* from *stream*'s output history, and ensures that all other output records that were covered by *record* are visible. In effect, this draws background ink over the record, and then redraws all the records that overlap *record*.

If *record* is not in the stream's output history, then an error is signalled, unless *errorp* is *false*.

If *record* is a list of output records, `erase-output-record` will delay the repainting operation until after all of the output records in *record* have been deleted. This is intended to be used by programmers as an optimization, since calling `erase-output-record` on a list of output records can be substantially faster than calling it repeatedly on each output record.

⇒ `copy-textual-output-history` *window stream* `&optional` *region record* [*Function*]

Given an *output recording stream window* and a character output stream *stream*, `copy-textual-output-history` maps over all of the textual output records in the region *region* and writes them to *stream*, in order from the top of the output to the bottom of the output.

If *record* is supplied, it is the top-level record to map over. It defaults to `stream-output-history` of *window*.

16.4.2 Graphics Output Recording

Using `draw-line*` as an example, calling any of the drawing functions specified in Section 12.5 and Section 12.7 on an output recording stream results in the following series of function calls:

- A program calls `draw-line*` on arguments *stream* (and output recording stream), *x1*, *y1*, *x2*, and *y2*, and perhaps some drawing options.
- `draw-line*` merges the supplied drawing options into the stream's medium, and then calls `medium-draw-line*` on the stream. (Note that a compiler macro could detect the case where there are no drawing options or constant drawing options, and do this at compile time.)
- The `:around` method for `medium-draw-line*` on the output recording stream is called. If `stream-recording-p` is *true*, this creates an output record with all of the information necessary to replay the output record. If `stream-drawing-p` is *true*, this then does a `call-next-method`. (Note that the `call-next-method` could be replaced by a call to the `medium-draw-line*` method on the stream's medium, avoiding the need for a trampolining function call.)
- An `:around` method for `medium-draw-line*` performs the necessary user transformations by applying the medium transformation to *x1*, *y1*, *x2*, and *y2*, and to the clipping region, and then calls the medium-specific method.
- The "real" `medium-draw-line*` transforms the start and end coordinates of the line by the stream's device transformation, decodes the ink and line style into port-specific objects,

and finally invokes a port-specific function (such as `xlib:draw-line`) to do the actual drawing.

`replay-output-record` for a graphics displayed output record simply binds the state of the medium to the state captured in the output record, and calls the medium drawing function (such as `medium-draw-line*`) directly on the medium, with `stream-recording-p` of the stream set to *false* and `stream-drawing-p` set to *true*.

16.4.3 Text Output Recording

Major issue: *This is the place where* `write-string` *and friends get captured in order to create output record. The generic functions include things like* `stream-write-string`*, which are specialized by output recording streams to do the output recording. Describe exactly what happens. — SWM*

⇒ `stream-text-output-record` *stream text-style* [*Generic Function*]

Returns a text output record for the *output recording stream stream* suitable for holding characters in the text style *text-style*. If there is a currently "open" text output record that can hold characters in the specified text style, it is simply returned. Otherwise a new text output record is created that can hold characters in that text style, and its starting cursor position set to the cursor position of *stream*.

⇒ `stream-close-text-output-record` *stream* [*Generic Function*]

Closes the *output recording stream stream*'s currently "open" text output record by recording the stream's current cursor position as the ending cursor position of the record and adding the text output record to *stream*'s current output record by calling `stream-add-output-record`.

If there is no "open" text output record, `stream-close-text-output-record` does nothing.

Calling `stream-finish-output` or `stream-force-output`, calling `redisplay`, setting the text cursor position (via `stream-set-cursor-position`, `terpri`, or `fresh-line`), creating a new output record (for example, via `with-new-output-record`), or changing the state of `stream-recording-p` must close the current text output record. Some CLIM implementations may also choose to close the current text output record when the stream's drawing options or text style are changed, depending on the exact implementation of text output records.

⇒ `stream-add-character-output` *stream character text-style width height baseline* [*Generic Function*]

Adds the character *character* to the *output recording stream stream*'s text output record in the *text style text-style*. *width* and *height* are the width and height of the character in device units. *baseline* is the new baseline for the stream. `stream-add-character-output` must be implemented by calling `add-character-output-to-text-record`.

`stream-write-char` on an output recording stream will call `stream-add-character-output` when `stream-recording-p` is *true*.

⇒ `stream-add-string-output` *stream string start end text-style width height baseline* [*Generic Function*]

Adds the string *string* to the *output recording stream stream*'s text output record in the *text style text-style*. *start* and *end* are integers that specify the substring within *string* to add to the text output record. *width* and *height* are the width and height of the string in device units. *baseline* is the new baseline for the stream. `stream-add-string-output` must be implemented by calling `add-string-output-to-text-record`.

`stream-write-string` on an output recording stream will call `stream-add-string-output` when `stream-recording-p` is *true*.

16.4.4 Output Recording Utilities

CLIM provides several helper macros to control the output recording facility.

⇒ `with-output-recording-options` *(stream* `&key` *record draw)* `&body` *body* [*Macro*]

Enables or disables output recording and/or drawing on the *output recording stream* designated by *stream*, within the extent of *body*.

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

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

⇒ `invoke-with-output-recording-options` *stream continuation record draw* [*Generic Function*]

Enables or disables output recording and/or drawing on the *output recording stream stream*, and calls the function *continuation* with the new output recording options in effect. *continuation* is a function of one argument, the stream; it has dynamic extent.

If *draw* is *false*, output to the stream is not drawn on the viewport, but recording proceeds according to *record*; if *draw* is *true*, the output is drawn. If *record* is `nil`, output recording is disabled, but output otherwise proceeds according to *draw*; if *draw* is *true*, output recording is enabled.

All output recording streams must implement a method for `invoke-with-output-recording-options`.

⇒ `with-new-output-record` *(stream* `&optional` *record-type record* `&rest` *initargs)* `&body` *body* [*Macro*]

Creates a new output record of type *record-type* (which defaults to `standard-sequence-output-record`) and then captures the output of *body* into the new output record, and inserts the new record into the current "open" output record associated with the *output recording stream* designated by *stream*. While *body* is being evaluated, the current output record for *stream* will be bound to the new output record.

If *record* is supplied, it is the name of a variable that will be lexically bound to the new output record inside of body. *initargs* are CLOS initargs that are passed to `make-instance` when the new output record is created.

`with-new-output-record` returns the output record it creates.

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

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

⇒ `invoke-with-new-output-record` *stream continuation record-type* `&rest` *initargs* `&key` *parent* `&allow-other-keys` [*Generic Function*]

Creates a new output record of type *record-type*. The function *continuation* is then called, and any output it does to the *output recording stream stream* is captured in the new output record. The new record is then inserted into the current "open" output record associated with *stream* (or the top-level output record if there is no currently "open" one). While *continuation* is being executed, the current output record for *stream* will be bound to the new output record.

continuation is a function of two arguments, the stream and the output record; it has dynamic extent.

initargs are CLOS initargs that are passed to `make-instance` when the new output record is created. The *parent* initarg is handled specially, and specifies what output record should serve as the parent for the newly created record. If unspecified, `stream-current-output-record` of *stream* will be used as the parent.

`invoke-with-new-output-record` returns the output record it creates.

All output recording streams must implement a method for `invoke-with-new-output-record`.

⇒ `with-output-to-output-record` *(stream* `&optional` *record-type record* `&rest` *initargs))* `&body` *body* [*Macro*]

This is identical to `with-new-output-record` except that the new output record is not inserted into the output record hierarchy, and the text cursor position of *stream* is initially bound to $(0, 0)$.

record-type is the type of output record to create, which defaults to `standard-sequence-output-`

`record`. *initargs* are CLOS initargs that are passed to `make-instance` when the new output record is created.

If *record* is supplied, it is a variable that will be bound to the new output record while body is evaluated.

`with-output-to-output-record` returns the output record it creates.

The *stream* argument is not evaluated, and must be a symbol that is bound to an output recording stream. If *stream* is `t`, `*standard-output*` is used. Unlike facilities such as `with-output-to-string`, *stream* must be an actual stream, but no output will be done to it. *body* may have zero or more declarations as its first forms.

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

⇒ `invoke-with-output-to-output-record` *stream continuation record-type* `&rest` *initargs* `&key` [*Generic Function*]

This is similar to `invoke-with-new-output-record` except that the new output record is not inserted into the output record hierarchy, and the text cursor position of *stream* is initially bound to $(0, 0)$. That is, when `invoke-with-output-to-output-record` is used, no drawing on the stream occurs and nothing is put into the stream's normal output history. The function *continuation* is called, and any output it does to *stream* is captured in the output record.

continuation is a function of two arguments, the stream and the output record; it has dynamic extent. *record-type* is the type of output record to create. *initargs* are CLOS initargs that are passed to `make-instance` when the new output record is created.

`invoke-with-output-to-output-record` returns the output record it creates.

All output recording streams must implement a method for `invoke-with-output-to-output-record`.

⇒ `make-design-from-output-record` *record* [*Generic Function*]

Makes a design that replays the *output record record* when drawn via `draw-design`. If *record* is changed after the design is made, the consequences are unspecified. Applying a transformation to the design and calling `draw-design` on the new design is equivalent to establishing the same transformation before creating the output record.

It is permissible for implementations to support this only for those output records that correspond to the geometric object classes (for example, the output records created by `draw-line*` and `draw-ellipse*`).

Chapter 17

Table Formatting

CLIM provides a mechanism for tabular formatting of arbitrary output.

To employ these facilities the programmer annotates some output-generating code with advisory macros that describe the high-level formatting constraints, for example, what parts of code produce a row of the table, what parts of that produce the cells in the row.

For example, the following produces a table consisting of three columns containing a number, its square, and its cube. The output can be seen in Figure 17.1.

```
(defun table-test (count stream)
  (fresh-line stream)
  (formatting-table (stream :x-spacing '(3 :character))
    (dotimes (i count)
      (formatting-row (stream)
        (formatting-cell (stream :align-x :right)
          (prin1 i stream))
        (formatting-cell (stream :align-x :right)
          (prin1 (* i i) stream))
        (formatting-cell (stream :align-x :right)
          (prin1 (* i i i) stream))))))
```

The general contract of these facilities is described in the next section.

17.1 Overview of Table Formatting Facilities

In general, table formatting involves a sharing of responsibilities between user-written code and CLIM code. Code that employs only the lower level output facilities has full control over "where every piece of ink goes" in the output. In contrast, code that employs CLIM's table formatting

```
0   0    0
1   1    1
2   4    8
3   9   27
4  16   64
5  25  125
6  36  216
7  49  343
8  64  512
9  81  729
```

Figure 17.1: Example of tabular output.

facilities passes control to CLIM at a higher level. The programmer benefits by being able to specify the appearance of output in more compact abstract terms, and by not having to write the code that constrains the output to appear in proper tabular form.

Tabular output consists of a rectangular array of pieces of output corresponding to the bounding rectangles of the output. Each piece of output forms the contents of a *table cell.* There is no restriction on the contents of a table cell; cells may contain text, graphics, even other tables. For purposes of this discussion, we draw a strong distinction between specifying what goes in a cell, and specifying how the cells are arranged to form a table.

Specifying the contents of a cell is the responsibility of the programmer. A programmer using the table formatting facilities can predict the appearance of any individual cell by simply looking at the code for that cell. A cell's appearance does not depend upon where in the table it lies, for instance. The only thing about a cell's appearance that cannot be predicted from that cell alone is the amount of space the table formatting has to introduce in order to perform the desired alignment.

Specifying the relative arrangements of cells to form a table is the responsibility of CLIM based on the advice of the programmer. The programmer advises CLIM about extra space to put between rows or columns, for instance, but does not directly control the absolute positioning of a cell's contents.

For purposes of understanding table formatting, the following model may be used.

- The code for a cell draws to a stream that has a "private" (local to that cell) drawing plane.
- After output for a cell has finished, the bounding rectangle of all output on the "private" drawing plane is found. The region within that bounding rectangle forms the contents of a cell.
- Additional rectangular regions, containing only background ink, are attached to the edges of the cell contents. These regions ensure that the cells satisfy the tabular constraints that within a row all cells have the same height, and within a column all cells have the same width. CLIM may also introduce additional background for other purposes as described below.

- The cells are assembled into rows and columns.

Some tables are "multiple column" tables, in which two or more rows of the table are placed side by side (usually with intervening spacing) rather than all rows being aligned vertically. Multiple column tables are generally used to produce a table that is more esthetically pleasing, or to make more efficient use of space on the output device. When a table is a multiple column table, one additional step takes place in the formatting of the table: the rows of the table are rearranged into multiple columns in which some rows are placed side by side.

The advice that the programmer gives to CLIM on how to assemble the table consists of the following:

- How to place the contents of the cell within the cell (such as centered vertically, flush-left, and so forth) The possibilities for this advice are described below.
- Optionally, how much additional space to insert between columns and between rows of the table.
- Optionally, whether to make all columns the same size.

The advice describing how to place the contents of the cell within the cell consists of two pieces—how to constrain the cell contents in the horizontal direction, and how to constrain them in the vertical direction.

17.2 Table Formatting Functions

⇒ `formatting-table` *(*`&optional` *stream* `&key` *x-spacing y-spacing multiple-columns multiple-columns-x-spacing equalize-column-widths (move-cursor* `t`*) record-type* `&allow-other-keys` *)* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be done in a tabular format. This must be used in conjunction with `formatting-row` or `formatting-column`, and `formatting-cell`. The table is placed so that its upper left corner is at the current text cursor position of *stream*. If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the last cell of the table.

The returned value is the output record corresponding to the table.

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

x-spacing specifies the number of units of spacing to be inserted between columns of the table; the default is the width of a space character in the current text style. *y-spacing* specifies the number of units of spacing to be inserted between rows in the table; the default is the default vertical spacing of the stream. Possible values for these two options option are:

- An integer—a size in the current units to be used for spacing.
- A string or character—the spacing is the width or height of the string or character in the current text style.
- A function—the spacing is the amount of horizontal or vertical space the function would consume when called on the stream.
- A list—the list is of the form `(`*number unit*`)`, where *unit* is one of `:point`, `:pixel`, `:mm`, `:character`, or `:line`. When *unit* is `:character`, the width of an "M" in the current text style is used as the width of one character.

multiple-columns is either `nil`, `t`, or an integer. If it is `t` or an integer, the table rows will be broken up into a multiple columns. If it is `t`, CLIM will determine the optimal number of columns. If it is an integer, it will be interpreted as the desired number of columns. *multiple-columns-x-spacing* has the same format as *x-spacing*. It controls the spacing between the multiple columns. It defaults to the value of the *x-spacing* option.

When the boolean *equalize-column-widths* is *true*, CLIM will make all of the columns have the same width (the width of the widest cell in any column in the entire table).

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

⇒ `formatting-row` *(*`&optional` *stream* `&key` *record-type* `&allow-other-keys` *)* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be grouped into a table row. All of the output performed by *body* becomes the contents of one row. This must be used inside of `formatting-table`, and in conjunction with `formatting-cell`.

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

Once a table has had a row added to it via `formatting-row`, no columns may be added to it.

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

⇒ `formatting-column` *(*`&optional` *stream* `&key` *record-type* `&allow-other-keys` *)* `&body` *body* [*Macro*]

Binds the local environment in such a way the output of *body* will be grouped into a table column. All of the output performed by *body* becomes the contents of one column. This must be used inside of `formatting-table`, and in conjunction with `formatting-cell`.

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

Once a table has had a column added to it via `formatting-column`, no rows may be added to it.

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

⇒ `formatting-cell` *(*`&optional` *stream* `&key` *(align-x* `':left`*) (align-y* `':baseline`*) min-width min-height record-type* `&allow-other-keys` *)* `&body` *body* *[Macro]*

Controls the output of a single cell inside a table row or column, or of a single item inside `formatting-item-list`. All of the output performed by *body* becomes the contents of the cell.

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

align-x specifies how the output in a cell will be aligned relative to other cells in the same table column. The default, `:left`, causes the cells to be flush-left in the column. The other possible values are `:right` (meaning flush-right in the column) and `:center` (meaning centered in the column). Each cell within a column may have a different alignment; thus it is possible, for example, to have centered legends over flush-right numeric data.

align-y specifies how the output in a cell will be aligned vertically. The default, `:baseline`, causes textual cells to be aligned along their baselines and graphical cells to be aligned at the bottom. The other possible values are `:bottom` (align at the bottom of the output), `:top` (align at the top of the output), and `:center` (center the output in the cell).

min-width and *min-height* are used to specify minimum width or height of the cell. The default, `nil`, causes the cell to be only as wide or high as is necessary to contain the cell's contents. Otherwise, *min-width* and *min-height* are specified in the same way as the `:x-spacing` and `:y-spacing` arguments to `formatting-table`.

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

⇒ `formatting-item-list` *(*`&optional` *stream* `&key` *x-spacing y-spacing n-columns n-rows stream-width stream-height max-width max-height initial-spacing (row-wise* `t`*) (move-cursor* `t`*) record-type* `&allow-other-keys` *)* `&body` *body* *[Macro]*

Binds the local environment in such a way that the output of *body* will be done in an item list (that is, menu) format. This must be used in conjunction with `formatting-cell`, which delimits each item. The item list is placed so that its upper left corner is at the current text cursor position of *stream*. If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the last cell of the item list.

"Item list output" is more strictly defined as: each row of the item list consists of a single cell. Rows are placed with the first row on top, and each succeeding row has its top aligned with the bottom of the previous row (plus the specified *y-spacing*). Multiple rows and columns are constructed after laying the item list out in a single column. Item list output takes place in a normalized $+y$-downward coordinate system.

If *row-wise* is *true* (the default) and the item list requires multiple columns, each successive element in the item list is layed out from left to right. If *row-wise* is *false* and the item list requires multiple columns, each successive element in the item list is layed out below its predecessor, like in a telephone book.

The returned value is the output record corresponding to the table.

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

x-spacing specifies the number of units of spacing to be inserted between columns of the item list; the default is the width of a `#\Space` character in the current text style. *y-spacing* specifies the number of units of spacing to be inserted between rows in the item list; the default is default vertical spacing of the stream. The format of these arguments is as for `formatting-table`.

When the boolean *equalize-column-widths* is *true*, CLIM will make all of the columns have the same width (the width of the widest cell in any column in the entire item list).

n-columns and *n-rows* specify the number of columns or rows in the item list. The default for both is `nil`, which causes CLIM to pick an aesthetically pleasing layout, possibly constrained by the other options. If both *n-columns* and *n-rows* are supplied and the item list contains more elements than will fit according to the specification, CLIM will format the item list as if *n-rows* were supplied as `nil`.

max-width and *max-height* constrain the layout of the item list. CLIM will not make the item list any wider than *max-width*, unless it is overridden by *n-rows*. It will not make the item list any taller than *max-height*, unless it is overridden by *n-columns*.

`formatting-item-list` normally spaces items across the entire width of the stream. When *initial-spacing* is *true*, it inserts some whitespace (about half as much space as is between each item) before the first item on each line. When it is *false* (the default), the initial whitespace is not inserted.

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

⇒ `format-items` *items* `&key` *stream printer presentation-type x-spacing y-spacing n-columns n-rows max-width max-height cell-align-x cell-align-y initial-spacing (row-wise* `t`*) (move-cursor* `t`*) record-type* [*Function*]

This is a function interface to the item list formatter. The elements of the sequence *items* are formatted as separate cells within the item list.

stream is an output recording stream to which output will be done. It defaults to `*standard-output*`.

printer must be a function that takes two arguments, an item and a stream, and outputs the item on the stream. *printer* has dynamic extent. The default for *printer* is `prin1`.

presentation-type is a presentation-type. When *printer* is not supplied, the items will be printed as if *printer* were

```
#'(lambda (item stream)
    (present item presentation-type :stream stream))
```

When *printer* is supplied, each item will be enclosed in a presentation whose type is *presentation-type*.

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

cell-align-x and *cell-align-y* are used to supply `:align-x` and `:align-y` to an implicitly used `formatting-cell`.

record-type is as for `formatting-item-list`.

17.3 The Table and Item List Formatting Protocols

Both table and item list formatting is implemented on top of the basic output recording protocol, using `with-new-output-record` to specify the appropriate type of output record. For example, `formatting-table` first collects all the output that belongs in the table into a collection of row, column, and cell output records, all of which are children of a single table output record. During this phase, `stream-drawing-p` is bound to `nil` and `stream-recording-p` is bound to `t`. When all the output has been generated, the table layout constraint solver (`adjust-table-cells` or `adjust-item-list-cells`) is called to compute the table layout, taking into account such factors as the widest cell in a given column. If the table is to be split into multiple columns, `adjust-multiple-columns` is now called. Finally, the table output record is positioned on the stream at the current text cursor position and then displayed by calling `replay` on the table (or item list) output record.

17.3.1 Table Formatting Protocol

Any output record class that implements the following generic functions is said to support the table formatting protocol.

In the following subsections, the term "non-table output records" will be used to mean any output record that is not a table, row, column, cell, or item list output record. When CLIM "skips over intervening non-table output records", this means that it will bypass all the output records between two such table output records (such as a table and a row, or a row and a cell) that are not records of those classes (most notably, presentation output records). CLIM implementations are encouraged to detect invalid nesting of table output records, such as a row within a row, a cell within a cell, or a row within a cell. Note that this does not prohibit the nesting of calls to `formatting-table`, it simply requires that programmers include the inner table within one of the cells of the outer table.

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

The protocol class that represents tabular output records; a subclass of `output-record`. If you want to create a new class that behaves like a table output record, it should be a subclass of `table-output-record`. All instantiable subclasses of `table-output-record` must obey the table output record protocol.

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

Returns *true* if *object* is a *table output record*, otherwise returns *false*.

⇒ `:x-spacing` [*Initarg*]
⇒ `:y-spacing` [*Initarg*]
⇒ `:multiple-columns-x-spacing` [*Initarg*]
⇒ `:equalize-column-widths` [*Initarg*]

All subclasses of `table-output-record` must handle these initargs, which are used to specify, respectively, the x and y spacing, the multiple column x spacing, and equal-width columns attributes of the table.

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

The instantiable class of output record that represents tabular output. Its children will be a sequence of either rows or columns, with presentation output records possibly intervening. This is a subclass of `table-output-record`.

⇒ `map-over-table-elements` *function table-record type* [*Generic Function*]

Applies *function* to all the rows or columns of *table-record* that are of type *type*. *type* is either `:row`, `:column`, or `:row-or-column`. *function* is a function of one argument, an output record; it has dynamic extent. `map-over-table-elements` is responsible for ensuring that rows, columns, and cells are properly nested. It must skip over intervening non-table output record structure, such as presentations.

`map-over-table-elements` returns `nil`.

⇒ `adjust-table-cells` *table-record stream* [*Generic Function*]

This function is called after the tabular output has been collected, but before it has been replayed. The method on `standard-table-output-record` implements the usual table layout constraint solver (described above) by moving the rows or columns of the table output record *table-record* and the cells within the rows or columns. *stream* is the stream on which the table is displayed.

⇒ `adjust-multiple-columns` *table-record stream* [*Generic Function*]

This is called after `adjust-table-cells` to account for the case where the programmer wants the table to have multiple columns. *table-record* and *stream* are as for `adjust-table-cells`.

17.3.2 Row and Column Formatting Protocol

Any output record class that implements the following generic functions is said to support the row (or column) formatting protocol.

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

The protocol class that represents one row in a table; a subclass of `output-record`. If you want to create a new class that behaves like a row output record, it should be a subclass of `row-output-record`. All instantiable subclasses of `row-output-record` must obey the row output record protocol.

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

Returns *true* if *object* is a *row output record*, otherwise returns *false*.

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

The instantiable class of output record that represents a row of output within a table. Its children will be a sequence of cells, and its parent (skipping intervening non-tabular records such as presentations) will be a table output record. This is a subclass of `row-output-record`.

⇒ `map-over-row-cells` *function row-record* [*Generic Function*]

Applies *function* to all the cells in the row *row-record*, skipping intervening non-table output record structure. *function* is a function of one argument, an output record corresponding to a table cell within the row; it has dynamic extent.

`map-over-row-cells` returns `nil`.

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

The protocol class that represents one column in a table; a subclass of `output-record`. If you want to create a new class that behaves like a column output record, it should be a subclass of `column-output-record`. All instantiable subclasses of `column-output-record` must obey the column output record protocol.

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

Returns *true* if *object* is a *column output record*, otherwise returns *false*.

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

The instantiable class of output record that represent a column of output within a table. Its children will be a sequence of cells, and its parent (skipping intervening non-tabular records such as presentations) will be a table output record; presentation output records may intervene. This is a subclass of `column-output-record`.

⇒ `map-over-row-cells` *function column-record* [*Generic Function*]

Applies *function* to all the cells in the column *column-record*, skipping intervening non-table output record structure. *function* is a function of one argument, an output record corresponding to a table cell within the column; it has dynamic extent.

`map-over-row-cells` returns `nil`.

17.3.3 Cell Formatting Protocol

Any output record class that implements the following generic functions is said to support the cell formatting protocol.

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

The protocol class that represents one cell in a table or an item list; a subclass of `output-record`. If you want to create a new class that behaves like a cell output record, it should be a subclass of `cell-output-record`. All instantiable subclasses of `cell-output-record` must obey the cell output record protocol.

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

Returns *true* if *object* is a *cell output record*, otherwise returns *false*.

⇒ `:align-x` [*Initarg*]
⇒ `:align-y` [*Initarg*]
⇒ `:min-width` [*Initarg*]
⇒ `:min-height` [*Initarg*]

All subclasses of `cell-output-record` must handle these initargs, which are used to specify, respectively, the x and y alignment, and the minimum width and height attributes of the cell.

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

The instantiable class of output record that represent a single piece of output within a table row or column, or an item list. Its children will either be presentations, or output records that represent displayed output. This is a subclass of `cell-output-record`.

⇒ `cell-align-x` *cell* [*Generic Function*]
⇒ `cell-align-y` *cell* [*Generic Function*]
⇒ `cell-min-width` *cell* [*Generic Function*]
⇒ `cell-min-height` *cell* [*Generic Function*]

These functions return, respectively, the x and y alignment and minimum width and height of the *cell output record cell*.

17.3.4 Item List Formatting Protocol

⇒ `item-list-output-record` [*Protocol Class*]

The protocol class that represents an item list; a subclass of `output-record`. If you want to create a new class that behaves like an item list output record, it should be a subclass of `item-list-output-record`. All instantiable subclasses of `item-list-output-record` must obey the item list output record protocol.

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

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

⇒ `:x-spacing` [*Initarg*]
⇒ `:y-spacing` [*Initarg*]
⇒ `:initial-spacing` [*Initarg*]
⇒ `:row-wise` [*Initarg*]
⇒ `:n-rows` [*Initarg*]
⇒ `:n-columns` [*Initarg*]
⇒ `:max-width` [*Initarg*]
⇒ `:max-height` [*Initarg*]

All subclasses of `item-list-output-record` must handle these initargs, which are used to specify, respectively, the x and y spacing, the initial spacing, row-wise, the desired number of rows and columns, and maximum width and height attributes of the item list.

⇒ `standard-item-list-output-record` [*Class*]

The instantiable output record that represents item list output. Its children will be a sequence of cells, with presentations possibly intervening. This is a subclass of `item-list-output-record`.

⇒ `map-over-item-list-cells` *function item-list-record* [*Generic Function*]

Applies *function* to all of the cells in *item-list-record*. `map-over-item-list-cells` must skip over intervening non-table output record structure, such as presentations. *function* is a function of one argument, an output record corresponding to a cell in the item list; it has dynamic extent.

`map-over-item-list-cells` returns `nil`.

⇒ `adjust-item-list-cells` *item-list-record stream* [*Generic Function*]

This function is called after the item list output has been collected, but before the record has been replayed. The method on `standard-item-list-output-record` implements the usual item list layout constraint solver. *item-list-record* is the item list output record, and *stream* is the stream on which the item list is displayed.

Chapter 18

Graph Formatting

CLIM provides a mechanism for arranging arbitrary output in a graph. The following code produces the graph shown in Figure 18.1.

```
(defun graph-test (stream &optional (orientation :horizontal))
  (fresh-line stream)
  (macrolet ((make-node (&key name children)
               `(list* ,name ,children)))
    (flet ((node-name (node)
             (car node))
           (node-children (node)
             (cdr node)))
      (let* ((2a (make-node :name "2A"))
             (2b (make-node :name "2B"))
             (2c (make-node :name "2C"))
             (1a (make-node :name "1A" :children (list 2a 2b)))
             (1b (make-node :name "1B" :children (list 2b 2c)))
             (root (make-node :name "0" :children (list 1a 1b))))
        (format-graph-from-roots
          (list root)
          #'(lambda (node s)
              (write-string (node-name node) s))
          #'node-children
          :orientation orientation
          :stream stream)))))
```

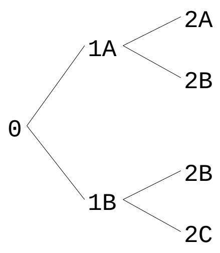


Figure 18.1: Example of graph formatting.

18.1 Graph Formatting Functions

⇒ `format-graph-from-roots` *root-objects object-printer inferior-producer* `&key` *stream orientation cutoff-depth merge-duplicates duplicate-key duplicate-test generation-separation within-generation-separation center-nodes arc-drawer arc-drawing-options graph-type (move-cursor* `t`*)* [*Function*]

Draws a graph whose roots are specified by the sequence *root-objects*. The nodes of the graph are displayed by calling the function *object-printer*, which takes two arguments, the node to display and a stream. *inferior-producer* is a function of one argument that is called on each node to produce a sequence of inferiors (or `nil` if there are none). Both *object-printer* and *inferior-producer* have dynamic extent.

The output from graph formatting takes place in a normalized $+y$-downward coordinate system. The graph is placed so that the upper left corner of its bounding rectangle is at the current text cursor position of *stream*. If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the graph.

The returned value is the output record corresponding to the graph.

stream is an output recording stream to which output will be done. It defaults to `*standard-output*`.

orientation may be either `:horizontal` (the default) or `:vertical`. It specifies which way the graph is oriented. CLIM implementations are permitted to extend the values of *orientation*, for example, adding `:right` or `:left` to distinguish between left-to-right or right-to-left layouts.

cutoff-depth specifies the maximum depth of the graph. It defaults to `nil`, meaning that there is no cutoff depth. Otherwise it must be an integer, meaning that no nodes deeper than *cutoff-depth* will be formatted or displayed.

If the boolean *merge-duplicates* is *true*, then duplicate objects in the graph will share the same node in the display of the graph. If *merge-duplicates* is *false* (the default), then duplicate objects will be displayed in separate nodes. (That is, when *merge-duplicates* is *false*, the resulting graph will be a tree.) *duplicate-key* is a function of one argument that is used to extract the node object component used for duplicate comparison; the default is `identity`. *duplicate-test* is a function of two arguments that is used to compare two objects to see if they are duplicates; the default is `eql`. *duplicate-key* and *duplicate-test* have dynamic extent.

generation-separation is the amount of space to leave between successive generations of the graph; the default should be chosen so that the resulting graph is visually pleasing. *within-generation-separation* is the amount of space to leave between nodes in the same generation of the graph; the default should be chosen so that the resulting graph is visually pleasing. *generation-separation* and *within-generation-separation* are specified in the same way as the *inter-row-spacing* argument to `formatting-table`.

When *center-nodes* is *true*, each node of the graph is centered with respect to the widest node in the same generation. The default is *false*.

arc-drawer is a function of seven positional and some unspecified keyword arguments that is responsible for drawing the arcs from one node to another; it has dynamic extent. The positional arguments are the stream, the "from" node's object, the "to" node's object, the "from" x and y position, and the "to" x and y position. The keyword arguments gotten from *arc-drawing-options* are typically line drawing options, such as for `draw-line*`. If *arc-drawer* is unsupplied, the default behavior is to draw a thin line from the "from" node to the "to" node using `draw-line*`.

graph-type is a keyword that specifies the type of graph to draw. All CLIM implementations must support graphs of type `:tree`, `:directed-graph` (and its synonym `:digraph`), and `:directed-acyclic-graph` (and its synonym `:dag`). *graph-type* defaults to `:digraph` when *merge-duplicates* is *true*, otherwise it defaults to `:tree`. Typically, different graph types will use different output record classes and layout engines to lay out the graph. However, it is permissible for all of the required graph types to use exactly the same graph layout engine.

18.2 The Graph Formatting Protocols

Graph formatting is implemented on top of the basic output recording protocol, using `with-new-output-record` to specify the appropriate type of output record. For example, `format-graph-from-roots` first collects all the output that belongs in the graph into a collection of graph node output records by calling `generate-graph-nodes`. All of the graph node output records are descendents of a single graph output record. During this phase, `stream-drawing-p` is bound to `nil` and `stream-recording-p` is bound to `t`. When all the output has been generated, the graph layout code (`layout-graph-nodes` and `layout-graph-edges`) is called to compute the graph layout. Finally, the graph output record is positioned on the stream at the current text cursor position and then displayed by calling `replay` on the graph output record.

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

The protocol class that represents a graph; a subclass of `output-record`. If you want to create a new class that behaves like a graph output record, it should be a subclass of `graph-output-record`. All instantiable subclasses of `graph-output-record` must obey the graph output record protocol.

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

Returns *true* if *object* is a *graph output record*, otherwise returns *false*.

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

The instantiable class of output record that represents a graph. Its children will be a sequence graph nodes. This is a subclass of `graph-output-record`.

⇒ `:orientation` [*Initarg*]
⇒ `:center-nodes` [*Initarg*]
⇒ `:cutoff-depth` [*Initarg*]
⇒ `:merge-duplicates` [*Initarg*]
⇒ `:generation-separation` [*Initarg*]
⇒ `:within-generation-separation` [*Initarg*]
⇒ `:hash-table` [*Initarg*]

All the graph output record must handle these seven initargs, which are used to specify, respectively, the orientation, node centering, cutoff depth, merge duplicates, generation and within-generation spacing, and the node hash table of a graph output record.

⇒ `define-graph-type` *graph-type class* [*Macro*]

Defines a new graph type named by the symbol *graph-type* that is implemented by the class *class*. *class* must be a subclass of `graph-output-record`. Neither of the arguments is evaluated.

All CLIM implementations must support graphs of type `:tree`, `:directed-graph` (and its synonym `:digraph`), and `:directed-acyclic-graph` (and its synonym `:dag`).

⇒ `graph-root-nodes` *graph-record* [*Generic Function*]

Returns a sequence of the graph node output records corresponding to the root objects for the graph output record *graph-record*.

⇒ `(setf graph-root-nodes)` *roots graph-record* [*Generic Function*]

Sets the root nodes of *graph-record* to *roots*.

⇒ `generate-graph-nodes` *graph-record stream root-objects object-printer inferior-producer* `&key` *duplicate-key duplicate-test* [*Generic Function*]

This function is responsible for generating all of the graph node output records of the graph. *graph-record* is the graph output record, and *stream* is the output stream. The graph node output records are generating by calling the object printer on the root objects, then (recursively) calling the inferior producer on the root objects and calling the object printer on all inferiors. After all of the graph node output records have been generated, the value of `graph-root-nodes` of *graph-record* must be set to be a sequence of the those graph node output records that correspond to the root objects.

root-objects, *object-printer*, *inferior-producer*, *duplicate-key*, and *duplicate-test* are as for `format-graph-from-roots`.

⇒ `layout-graph-nodes` *graph-record stream arc-drawer arc-drawing-options* [*Generic Function*]

This function is responsible for laying out the nodes in the graph contained in the output record *graph-record.* It is called after the graph output has been collected, but before the graph record has been displayed. The method on `standard-graph-output-record` implements the usual graph layout constraint solver. *stream* is the stream on which the graph is displayed.

⇒ `layout-graph-edges` *graph-record stream arc-drawer arc-drawing-options* [*Generic Function*]

This function is responsible for laying out the edges in the graph. It is called after the graph nodes have been layed out, but before the graph record has been displayed. The method on `standard-graph-output-record` simply causes thin lines to be drawn from each node to all of its children. *graph-record* and *stream* are as for `layout-graph-nodes`.

⇒ `graph-node-output-record` [*Protocol Class*]

The protocol class that represents a node in graph; a subclass of `output-record`. If you want to create a new class that behaves like a graph node output record, it should be a subclass of `graph-node-output-record`. All instantiable subclasses of `graph-node-output-record` must obey the graph node output record protocol.

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

Returns *true* if *object* is a *graph node output record*, otherwise returns *false*.

⇒ `standard-graph-node-output-record` [*Class*]

The instantiable class of output record that represents a graph node. Its parent will be a graph output record. This is a subclass of `graph-node-output-record`.

⇒ `graph-node-parents` *graph-node-record* [*Generic Function*]

Returns a sequence of the graph node output records whose objects are "parents" of the object corresponding to the graph node output record *graph-node-record.* Note that this is not the same as `output-record-parent`, since `graph-node-parents` can return output records that are not the parent records of *graph-node-record.*

⇒ `(setf graph-node-parents)` *parents graph-node-record* [*Generic Function*]

Sets the parents of *graph-node-record* to be *parents. parents* must be a list of graph node records.

⇒ `graph-node-children` *graph-node-record* [*Generic Function*]

Returns a sequence of the graph node output records whose objects are "children" of the object corresponding to the graph node output record *graph-node-record.* Note that this is not the same as `output-record-children`, since `graph-node-children` can return output records that are not child records of *graph-node-record.*

⇒ `(setf graph-node-children)` *children graph-node-record* [*Generic Function*]

Sets the children of *graph-node-record* to be *children. children* must be a list of graph node records.

⇒ `graph-node-object` *graph-node-record* [*Generic Function*]

Returns the object that corresponds to the graph node output record *graph-node-record*. It is permissible for this function to work correctly only while inside of the call to **`format-graph-from-roots`**. It is unspecified what result will be returned outside of **`format-graph-from-roots`**. This restriction is permitted so that CLIM is not required to capture application objects that might have dynamic extent.

Chapter 19

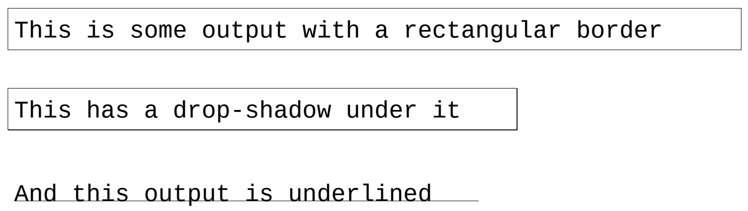
Bordered Output

CLIM provides a mechanism for surrounding arbitrary output with some kind of a border. The programmer annotates some output-generating code with an advisory macro that describes the type of border to be drawn. The following code produces the output shown in Figure 19.1.

For example, the following produces three pieces of output, surrounded by a rectangular, highlighted with a dropshadow, and underlined, respectively.

```
(defun border-test (stream)
  (fresh-line stream)
  (surrounding-output-with-border (stream :shape :rectangle)
    (format stream "This is some output with a rectangular border"))
  (terpri stream) (terpri stream)
  (surrounding-output-with-border (stream :shape :drop-shadow)
    (format stream "This has a drop-shadow under it"))
  (terpri stream) (terpri stream)
  (surrounding-output-with-border (stream :shape :underline)
    (format stream "And this output is underlined")))
```

⇒ `surrounding-output-with-border` *(*`&optional` *stream* `&rest` *drawing-options* `&key` *shape (move-cursor* `t`*))* `&body` *body* [*Macro*]



```
This is some output with a rectangular border

This has a drop-shadow under it

And this output is underlined
```

Figure 19.1: Examples of bordered output.

Binds the local environment in such a way the output of *body* will be surrounded by a border of the specified shape. Every implementation must support the shapes `:rectangle` (the default), `:oval`, `:drop-shadow`, and `:underline`. `:rectangle` draws a rectangle around the bounding rectangle of the output. `:oval` draws an oval around the bounding rectangle of the output. `:drop-shadow` draws a "drop shadow" around the lower right edge of the bounding rectangle of the output. `:underline` draws a thin line along the baseline of all of the text in the output, but does not draw anything underneath non-textual output. *drawing-options* is a list of drawing options that are passed to the function that draws the border.

If the boolean *move-cursor* is *true* (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the bordered output.

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

There are several strategies for implementing borders. One strategy is to create a "border output record" that contains the output records produced by the output of *body*, plus one or more output records that represent the border. Another strategy might be to arrange to call the border drawer at the approriate times without explicitly recording it.

⇒ `define-border-type` *shape arglist* `&body` *body* [*Macro*]

Defines a new kind of border named *shape*. *arglist* must be a subset of the "canonical" arglist below (using `string-equal` to do the comparison):
(`&key` *stream record left top right bottom)*

arglist may include other keyword arguments that serve as the drawing options.

body is the code that actually draws the border. It has lexical access to `stream`, `record`, `left`, `top`, `right`, and `bottom`, which are respectively, the stream being drawn on, the output record being surrounded, and the coordinates of the left, top, right, and bottom edges of the bounding rectangle of the record. *body* may have zero or more declarations as its first forms.

Chapter 20

Text Formatting

20.1 Textual List Formatting

⇒ `format-textual-list` *sequence printer* `&key` *stream separator conjunction* [*Function*]

Outputs the sequence of items in *sequence* as a "textual list". For example, the list `(1 2 3 4)` might be printed as

```
1, 2, 3, and 4
```

printer is a function of two arguments: an element of the sequence and a stream; it has dynamic extent. It is called to output each element of the sequence.

stream specifies the output stream. The default is `*standard-output*`.

The *separator* and *conjunction* arguments provide control over the appearance of each element of the sequence and over the separators used between each pair of elements. *separator* is a string that is output after every element but the last one; the default for *separator* is `", "` (that is, a comma followed by a space). *conjunction* is a string that is output before the last element. The default is `nil`, meaning that there is no conjunction. Typical values for *conjunction* are the strings `"and"` and `"or"`.

20.2 Indented Output

⇒ `indenting-output` *(stream indentation* `&key` *(move-cursor* `t`*))* `&body` *body* [*Macro*]

Binds *stream* to a stream that inserts whitespace at the beginning of each line of output produced by *body*, and then writes the indented output to the stream that is the original value of *stream*.

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

indentation specifies how much whitespace should be inserted at the beginning of each line. It is specified in the same way as the `:x-spacing` option to `formatting-table`.

If the boolean *move-cursor* is *true* (the default), CLIM moves the cursor to the end of the table.

Programmers using `indenting-output` should begin the body with a call to *fresh-line* (or some equivalent) to position the stream to the initial indentation.

Implementation note: Some CLIM implementations restrict the use of `indenting-output` and `filling-output` such that a call to `indenting-output` should appear outside of a call to `filling-output`. Implementations are encouraged to relax this restriction if the behavior is well-defined, but uses of `indenting-output` inside of `filling-output` may not be portable.

20.3 Filled Output

⇒ `filling-output` *(stream* `&key` *fill-width break-characters after-line-break after-line-break-initially)* `&body` *body* [*Macro*]

Binds *stream* to a stream that inserts line breaks into the textual output written to it (by such functions as `write-char` and `write-string`) so that the output is usually no wider then *fill-width*. The filled output is then written on the original stream.

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

fill-width specifies the width of filled lines, and defaults to 80 characters. It is specified the same way as the `:x-spacing` option for `formatting-table`.

"Words" are separated by the characters specified in the list *break-characters*. When a line is broken to prevent wrapping past the end of a line, the line break is made at one of these separators. That is, `filling-output` does not split "words" across lines, so it might produce output wider than *fill-width*.

after-line-break specifies a string to be sent to *stream* after line breaks; the string appears at the beginning of each new line. The string must not be wider than *fill-width*.

If the boolean *after-line-break-initially* is *true*, then the *after-line-break* text is to be written to *stream* before executing *body*, that is, at the beginning of the first line. The default is *false*.