# Pict: Functional Pictures

Version 9.2.900

July 30, 2026

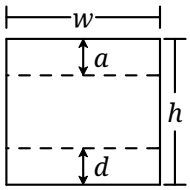
```
(require pict)
```
package: `pict-lib`

The `pict` library is one of the standard Racket functional picture libraries (the other being `2htdp/image`). This library was originally designed for use with Slideshow, and is re-provided by the `slideshow` language.

# 1 Pict Datatype

A *pict* is a `pict` structure representing an image. Some functions, such as `hline`, create new simple picts. Other functions, such as `ht-append`, build new picts out of existing picts. In the latter case, the embedded picts retain their identity, so that offset-finding functions, such as `lt-find`, can find the offset of an embedded pict in a larger pict.

In addition to its drawing part, a pict has the following *bounding box* structure:




That is, the bounding box has a width $w$ and a height $h$. For a single text line, $d$ is descent below the baseline, and $a+d=h$. For multiple text lines (often created with a function like `vc-append`), $a$ is the ascent of the top line, and $d$ is the descent of the bottom line, so $a+d<h$. Many picts have $d=0$ and $a=h$.

In addition, a pict can have a *last* sub-pict that corresponds to the last item on the last line of text, so that extra lines can be added to the last line. In particular, the *last* element is useful for adding closing parentheses to a block of Racket code, where the last line of code not the longest line in the block.

The size information for a pict is computed when the pict is created. This strategy supports programs that create new picts though arbitrarily complex computations on the size and shape of existing picts. The functions `pict-width`, `pict-height`, `pict-descent`, and `pict-ascent` extract bounding box information from a pict.

A pict is a convertible datatype through the `file/convertible` protocol. Supported conversions include `'png-bytes`, `'eps-bytes`, `'pdf-bytes`, `'svg-bytes`, and variants such as `'png-bytes+bounds` and `'png-bytes+bounds8`.

A pict is serializable via `racket/serialize`, but serialization loses sub-pict information (preserving only the pict's drawing and bounding box).

All of the pict functions that accept picts also accept values that are pict convertible, meaning that picts can be mixed and matched with values from various other libraries.

Changed in version 1.2 of package `pict-lib`: Added support for `'png-bytes+bounds`, `'png-bytes+bounds8` and similar variants.
Changed in version 1.3: Enabled serialization.

```
(struct pict (draw
              width
              height
              ascent
              descent
              children
              panbox
              last)
    #:extra-constructor-name make-pict)
  draw : any/c
  width : real?
  height : real?
  ascent : real?
  descent : real?
  children : (listof child?)
  panbox : (or/c #f any/c)
  last : (or/c #f pict-path?)
```

A `pict` structure is normally not created directly with `make-pict`. Instead, functions like `text`, `hline`, and `dc` are used to construct a pict.

The `draw` field contains the pict's drawing information in an internal format. Roughly, the drawing information is a procedure that takes a `dc<%>` drawing context and an offset for the pict's top-left corner (i.e., it's bounding box's top left corner relative to the `dc<%>` origin). The state of the `dc<%>` is intended to affect the pict's drawing; for example, the pen and brush will be set for a suitable default drawing mode, and the `dc<%>` scale will be set to scale the resulting image. Use `draw-pict` (as opposed to `pict-draw`) to draw the picture.

The `panbox` field is internal and initialized to `#f`.

The `last` field indicates a pict within the `children` list (transitively) that can be treated as the last element of the last line in the pict. A `#f` value means that the pict is its own last sub-pict.

```
(struct child (pict dx dy sx sy syx sxy)
    #:extra-constructor-name make-child)
  pict : pict?
  dx : real?
  dy : real?
  sx : real?
  sy : real?
  syx : real?
  sxy : real?
```

Records, for a pict constructed of other picts, the transformation to arrive at a inverted point in the composed pict from an inverted point in a constituent pict's. An *inverted point* is a

point relative to a pict's lower-left corner with an increasing value moving upward.

A `child` structure is normally not created directly with `make-child`. Instead, functions like `hc-append` create `child` structures when combining picts to create a new one.

```
(explain  p
          [#:border bord
           #:ascent asc
           #:baseline base
           #:scale s
           #:line-width lw]) → pict?
  p : pict-convertible?
  bord : (or/c #f string? (is-a?/c color%)) = "firered"
  asc : (or/c #f string? (is-a?/c color%)) = "seagreen"
  base : (or/c #f string? (is-a?/c color%)) = "royalblue"
  s : real? = 5
  lw : real? = 1
```

Draw a `pict` like *p*, but the pict's bounding box is framed in the color *bord*, the baseline is draw in *base*, and the line defined by the ascent is drawn in *asc*. If any are `#f` then that part is not drawn. The pict is scaled up by *s*. The lines drawn are of width *lw* before scaling is applied.

Note that for single lines of text the baseline and ascent are the same, so only one line will be visible.

Examples:

```
> (define t (text "ij"))
> (define tt (vl-append t t))
> (explain t)
```

```
> (explain tt)
```

Added in version 1.10 of package `pict-lib`.

```
(explain-child  p
                path ...
               [#:border bord
                #:ascent asc
                #:baseline base
                #:scale s
                #:line-width lw]) → pict?
  p : pict-convertible?
  path : pict-path?
  bord : (or/c #f string? (is-a?/c color%)) = "firered"
  asc : (or/c #f string? (is-a?/c color%)) = "seagreen"
  base : (or/c #f string? (is-a?/c color%)) = "royalblue"
  s : real? = 5
  lw : real? = 1
```

Like `explain`, except the the explanation is drawn for some each inner `pict` of *p*, pointed to by *path*.

Examples:

```
> (define t1 (text "ij"))
> (define t2 (text "ij"))
> (define t3 (vl-append t1 t2))
> (explain-child t3 t3)
```

```
> (explain-child t3 t1)
```

```
> (explain-child t3 t2)
```

```
> (explain-child t3 t1 t2)
```

Added in version 1.10 of package `pict-lib`.

## 2 Basic Pict Constructors

```
(dc draw w h [a d]) → pict?
  draw : (-> (is-a?/c dc<%>) real? real? any)
  w : real?
  h : real?
  a : real? = h
  d : real? = 0
```

Creates an arbitrary self-rendering pict. The arguments to the rendering procedure will be a drawing context and top-left location for drawing.

The *w*, *h*, *a*, and *d* arguments determine the width, height, ascent, and descent of the of the resulting pict's bounding box respectively.

When the rendering procedure is called, the current pen and brush will be `'solid` and in the pict's color and `linewidth`, and the scale and offset of the drawing context will be set. The text mode will be transparent, but the font and text colors are not guaranteed to be anything in particular.

Example:

```
> (dc (λ (dc dx dy)
        (define old-brush (send dc get-brush))
        (define old-pen (send dc get-pen))
        (send dc set-brush
          (new brush% [style 'fdiagonal-hatch]
                      [color "darkslategray"]))
        (send dc set-pen
          (new pen% [width 3] [color "slategray"]))
        (define path (new dc-path%))
        (send path move-to 0 0)
        (send path line-to 50 0)
        (send path line-to 25 50)
        (send path close)
        (send dc draw-path path dx dy)
        (send dc set-brush old-brush)
        (send dc set-pen old-pen))
      50 50)
```

The *draw* is called during the dynamic extent of the call to `dc` as part of the contract checking.

Specifically, the pre-condition portion of the contract for dc concocts a dc<%> object with a random initial state, calls the *draw* argument with that dc<%> and then checks to make sure that *draw* the state of the dc<%> object is the same as it was before *draw* was called.

Example:

```
> (dc (λ (dc dx dy)
        (send dc set-brush "red" 'solid)
        (send dc set-pen "black" 1 'transparent)
        (send dc draw-ellipse dx dy 50 50))
      50 50)
dc: contract violation;
 draw proc does not restore the dc state after being called
       draw: #<procedure>
  in: (->i
       ((draw
         (-> (is-a?/c dc<%>) real? real? any))
        (w real?)
        (h real?))
       ((d (or/c #f real?)) (a (or/c #f real?)))
       #:pre/name
       (draw)
       "draw proc does not restore the dc state after being
called"
       (... draw)
       (p pict?))
  contract from: <pkgs>/pict-lib/pict/main.rkt
  blaming: top-level
   (assuming the contract is correct)
  at: <pkgs>/pict-lib/pict/main.rkt:101:3
```

Changed in version 1.3 of package `pict-lib`: The *draw* argument is now called by the `#:pre` condition of dc.

```
(unsafe-dc draw w h [a d]) → pict?
  draw : (-> (is-a?/c dc<%>) real? real? any)
  w : real?
  h : real?
  a : real? = h
  d : real? = 0
```

Like dc, except that the *draw* procedure is not called during the precondition contract checking.

```
(blank [size]) → pict?
  size : real? = 0
(blank w h) → pict?
```

```
  w : real?
  h : real?
(blank w a d) → pict?
  w : real?
  a : real?
  d : real?
(blank w h a d) → pict?
  w : real?
  h : real?
  a : real?
  d : real?
```

Creates a pict that draws nothing. The one-argument case supplies a value used for both the width and height of the resulting pict's bounding box. In the one- and two-argument case, the ascent and descent are 0 for the resulting pict's bounding box; in the three-argument case, the height is computed by adding the given ascent and descent.

Example:

```
> (blank 50)
```

```
(text content [style size angle]) → pict?
  content : string?
  style : text-style/c = null
  size : (integer-in 1 1024) = 12
  angle : real? = 0
```

Creates a pict that draws text. For creating text picts within a slide presentation, see `t`. The size of the resulting pict may depend on the value of `dc-for-text-size`.

The *style* argument must be one of the following:

- `null` — the default, same as `'default`
- a `font%` object
- a font family symbol, such a `'roman` (see `font%`)
- a font face string, such as `"Helvetica"` (see `font%`)
- `(cons str sym)` combining a face string and a font family (in case the face is unavailable; see `font%`)

- `(cons 'bold style)` for a valid *style*
- `(cons (cons 'weight weight) style)` where *weight* is a font weight

  Added in version 1.14 of package `pict-lib`.
- `(cons 'italic style)`
- `(cons 'subscript style)`
- `(cons 'superscript style)`
- `(cons 'large-script style)` — makes subscripts and superscripts larger, which is more suitable for small text sizes as might appear in print

  Added in version 1.5 of package `pict-lib`.
- `(cons 'caps style)`
- `(cons 'combine style)` — allows kerning and ligatures (the default, unless the `'modern` family is specified)
- `(cons 'no-combine style)` — renders characters individually
- `(cons 'aligned style)` — enables hinting, which rounds metrics to integers
- `(cons 'unaligned style)` — disables hinting (which is the default), so that metrics are scalable
- `(cons color style)` — where *color* is a `color%` object, `colorize`s the text

If both `'combine` and `'no-combine` are specified, the first one in *style* takes precedence. Similarly, if both `'aligned` and `'unaligned` are specified, the first one in *style* takes precedence. If `'caps` is specified, the *angle* must be zero.

The given *size* is in pixels, but it is ignored if a `font%` object is provided in the text-style.

The *angle* is in radians, and positive values rotate counter-clockwise. For a non-zero *angle*, the resulting pict's bounding box covers the rotated text, and the descent is zero and the ascent is the height.

Examples:

```
> (text "tom collins")
```

tom collins

```
> (text "g & t" (cons 'bold 'roman))
```

**g & t**

```
> (text "martini" null 13 (/ pi 2))
```

martini

```
(hline w h [#:segment seg-length]) → pict?
  w : real?
  h : real?
  seg-length : (or/c #f real?) = #f
(vline w h [#:segment seg-length]) → pict?
  w : real?
  h : real?
  seg-length : (or/c #f real?) = #f
```

Straight lines, centered within their bounding boxes.

Examples:

```
> (hline 40 5)
```



```
> (hline 40 30)
```



```
> (vline 5 40 #:segment 5)
```



Changed in version 1.11 of package `pict-lib`: Removed implicit truncation of some coordinates to integers.

```
(frame pict
       [#:segment seg-length
        #:color color
        #:line-width width]) → pict?
  pict : pict-convertible?
  seg-length : (or/c #f real?) = #f
  color : (or/c #f string? (is-a?/c color%)) = #f
  width : (or/c #f real?) = #f
```

Frames a given pict. If the color or line width are provided, they override settings supplied by the context.

Examples:

```
> (frame (circle 30))
```

```
> (frame (circle 30) #:segment 5)
```

```
> (frame (circle 30) #:color "chartreuse" #:line-width 3)
```

```
(ellipse  w
          h
         [#:border-color border-color
          #:border-width border-width]) → pict?
  w : real?
  h : real?
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
(circle  diameter
        [#:border-color border-color
         #:border-width border-width]) → pict?
  diameter : real?
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
(filled-ellipse  w
                 h
                [#:draw-border? draw-border?
                 #:color color
                 #:border-color border-color
                 #:border-width border-width]) → pict?
  w : real?
  h : real?
  draw-border? : any/c = #t
  color : (or/c #f string? (is-a?/c color%)) = #f
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
(disk  diameter
      [#:draw-border? draw-border?
       #:color color
       #:border-color border-color
       #:border-width border-width]) → pict?
  diameter : (and/c rational? (not/c negative?))
  draw-border? : any/c = #t
  color : (or/c #f string? (is-a?/c color%)) = #f
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
```

Unfilled and filled ellipses.

If *draw-border?* is #f, then the pen is set to be transparent before drawing the ellipse. The *color*, *border-color* and *border-width* arguments control the fill color, color of the

border, and width of the border, respectively. If these arguments are `#f`, values set using `linewidth` and `colorize` are used instead. Passing non-`#f` values as *`border-color`* or *`border-width`* when *`draw-border?`* is `#f` results in a contract violation.

Examples:

```
> (ellipse 40 30)
```

```
> (circle 30)
```

```
> (filled-ellipse 30 40)
```

```
> (disk 30)
```

```
> (disk 40 #:color "Chartreuse" #:border-color "Medium Aquama-
rine" #:border-width 5)
```

Changed in version 1.4 of package `pict-lib`: Added `#:color`, `#:border-color` and `#:border-width` arguments.

```
(rectangle  w
            h
           [#:border-color border-color
            #:border-width border-width]) → pict?
  w : real?
  h : real?
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
(filled-rectangle  w
                   h
                  [#:draw-border? draw-border?
                   #:color color
                   #:border-color border-color
                   #:border-width border-width]) → pict?
  w : real?
  h : real?
  draw-border? : any/c = #t
  color : (or/c #f string? (is-a?/c color%)) = #f
```

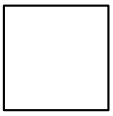
```
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
```

Unfilled and filled rectangles.

If *draw-border?* is #f, then the pen is set to be transparent before drawing the rectangle. The *color*, *border-color* and *border-width* arguments control the fill color, color of the border, and width of the border, respectively. If these arguments are #f, values set using linewidth and colorize are used instead. Passing non-#f values as *border-color* or *border-width* when *draw-border?* is #f results in a contract violation.

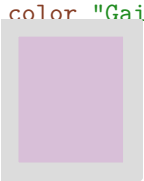
Examples:

```
> (rectangle 50 50)
```



```
> (filled-rectangle 50 80)
```

```
> (filled-rectangle 60 70 #:color "Thistle" #:border-
color "Gainsboro" #:border-width 10)
```



Changed in version 1.4 of package `pict-lib`: Added `#:color`, `#:border-color` and `#:border-width` arguments.

```
(rounded-rectangle  w
                    h
                   [corner-radius
                    #:angle angle
                    #:border-color border-color
                    #:border-width border-width]) → pict?
  w : real?
  h : real?
  corner-radius : real? = -0.25
  angle : real? = 0
  border-color : (or/c #f string? (is-a?/c color%)) = #f
```

```
  border-width : (or/c #f real?) = #f
(filled-rounded-rectangle  w
                           h
                          [corner-radius
                           #:angle angle
                           #:draw-border? draw-border?
                           #:color color
                           #:border-color border-color
                           #:border-width border-width]) → pict?
  w : real?
  h : real?
  corner-radius : real? = -0.25
  angle : real? = 0
  draw-border? : any/c = #t
  color : (or/c #f string? (is-a?/c color%)) = #f
  border-color : (or/c #f string? (is-a?/c color%)) = #f
  border-width : (or/c #f real?) = #f
```

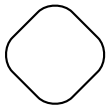
Unfilled and filled rectangles with rounded corners. The *corner-radius* is used to determine how much rounding occurs in the corners. If it is a positive number, then it determines the radius of a circle touching the edges in each corner, and the rounding of the rectangle follow the edge of those circles. If it is a negative number, then the radius of the circles in the corners is the absolute value of the *corner-radius* times the smaller of *w* and *h*.

The *angle* determines how much the rectangle is rotated, in radians.

If *draw-border?* is #f, then the pen is set to be transparent before drawing the rectangle. The *color*, *border-color* and *border-width* arguments control the fill color, color of the border, and width of the border, respectively. If these arguments are #f, values set using linewidth and colorize are used instead. Passing non-#f values as *border-color* or *border-width* when *draw-border?* is #f results in a contract violation.

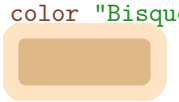
Examples:

```
> (rounded-rectangle 40 40 -0.3 #:angle (/ pi 4))
```



```
> (filled-rounded-rectangle 50 40)
```

```
> (filled-rounded-rectangle 70 30 #:color "Burlywood" #:border-
color "Bisque" #:border-width 8)
```



Changed in version 1.4 of package `pict-lib`: Added `#:color`, `#:border-color` and `#:border-width` arguments.

```
(bitmap img) → pict
  img : (or/c path-string?
              (is-a?/c bitmap%)
              pict-convertible?
              convertible?)
```

Returns a pict that displays a bitmap. When a path is provided, the image is loaded with the `'unknown/mask` flag, which means that a mask bitmap is generated if the file contains a mask.

If the bitmap cannot be loaded, if the given `bitmap%` object is not valid, or if the `bitmap-draft-mode` parameter is set to `#t`, the result pict draws the words “bitmap failed”.

If *img* is both convertible and pict convertible, then the pict conversion is used. If both apply, the pict conversion is used. If pict conversion is used, the pict is drawn into a bitmap and the result of `bitmap` draws that bitmap. If convertible is used, it is used with the `'png-bytes` conversion mode.

```
(arrow size radians) → pict?
  size : real?
  radians : real?
(arrowhead size radians) → pict?
  size : real?
  radians : real?
```

Creates an arrow or arrowhead in the specific direction within a *size* by *size* pict. Points on the arrow may extend slightly beyond the bounding box.

Examples:

```
> (arrow 30 0)
```

```
> (arrow 30 (/ pi 2))
```

```
> (arrowhead 30 0)
```

```
(pip-line dx dy size) → pict?
  dx : real?
  dy : real?
```

```
  size : real?
(pip-arrow-line dx dy size) → pict?
  dx : real?
  dy : real?
  size : real?
(pip-arrows-line dx dy size) → pict?
  dx : real?
  dy : real?
  size : real?
```

Creates a line (with some number of arrowheads) as a zero-sized pict suitable for use with `pin-over`. The 0-sized picture contains the starting point.

The *size* is used for the arrowhead size. Even though `pip-line` creates no arrowheads, it accepts the *size* argument for consistency with the other functions.

```
(pin-line  pict
           src
           find-src
           dest
           find-dest
          [#:start-angle start-angle
           #:end-angle end-angle
           #:start-pull start-pull
           #:end-pull end-pull
           #:line-width line-width
           #:color color
           #:alpha alpha
           #:style style
           #:under? under?
           #:label label
           #:x-adjust-label x-adjust-label
           #:y-adjust-label y-adjust-label]) → pict?
  pict : pict-convertible?
  src : pict-path?
  find-src : (pict-convertible? pict-path? . -> . (values real? real?))
  dest : pict-path?
  find-dest : (pict-convertible? pict-path? . -> . (values real? real?))
  start-angle : (or/c real? #f) = #f
  end-angle : (or/c real? #f) = #f
  start-pull : real? = 1/4
  end-pull : real? = 1/4
  line-width : (or/c #f real?) = #f
  color : (or/c #f string? (is-a?/c color%)) = #f
  alpha : (real-in 0.0 1.0) = 1.0
```

```
  style : (or/c 'transparent 'solid 'xor 'hilite        = #f
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash #f)
  under? : any/c = #f
  label : pict? = (blank)
  x-adjust-label : real? = 0
  y-adjust-label : real? = 0
(pin-arrow-line  arrow-size
                 pict
                 src
                 find-src
                 dest
                 find-dest
                [#:start-angle start-angle
                 #:end-angle end-angle
                 #:start-pull start-pull
                 #:end-pull end-pull
                 #:line-width line-width
                 #:color color
                 #:alpha alpha
                 #:style style
                 #:under? under?
                 #:label label
                 #:x-adjust-label x-adjust-label
                 #:y-adjust-label y-adjust-label
                 #:solid? solid?
                 #:hide-arrowhead? hide-arrowhead?]) → pict?
  arrow-size : real?
  pict : pict-convertible?
  src : pict-path?
  find-src : (pict-convertible? pict-path? . -> . (values real? real?))
  dest : pict-path?
  find-dest : (pict-convertible? pict-path? . -> . (values real? real?))
  start-angle : (or/c real? #f) = #f
  end-angle : (or/c real? #f) = #f
  start-pull : real? = 1/4
  end-pull : real? = 1/4
  line-width : (or/c #f real?) = #f
  color : (or/c #f string? (is-a?/c color%)) = #f
  alpha : (real-in 0.0 1.0) = 1.0
  style : (or/c 'transparent 'solid 'xor 'hilite        = #f
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash #f)
  under? : any/c = #f
```

```
  label : pict? = (blank)
  x-adjust-label : real? = 0
  y-adjust-label : real? = 0
  solid? : any/c = #t
  hide-arrowhead? : any/c = #f
(pin-arrows-line  arrow-size
                  pict
                  src
                  find-src
                  dest
                  find-dest
                 [#:start-angle start-angle
                  #:end-angle end-angle
                  #:start-pull start-pull
                  #:end-pull end-pull
                  #:line-width line-width
                  #:color color
                  #:alpha alpha
                  #:style style
                  #:under? under?
                  #:label label
                  #:x-adjust-label x-adjust-label
                  #:y-adjust-label y-adjust-label
                  #:solid? solid?
                  #:hide-arrowhead? hide-arrowhead?]) → pict?
  arrow-size : real?
  pict : pict-convertible?
  src : pict-path?
  find-src : (pict-convertible? pict-path? . -> . (values real? real?))
  dest : pict-path?
  find-dest : (pict-convertible? pict-path? . -> . (values real? real?))
  start-angle : (or/c real? #f) = #f
  end-angle : (or/c real? #f) = #f
  start-pull : real? = 1/4
  end-pull : real? = 1/4
  line-width : (or/c #f real?) = #f
  color : (or/c #f string? (is-a?/c color%)) = #f
  alpha : (real-in 0.0 1.0) = 1.0
  style : (or/c 'transparent 'solid 'xor 'hilite        = #f
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash #f)
  under? : any/c = #f
  label : pict-convertible? = (blank)
  x-adjust-label : real? = 0
  y-adjust-label : real? = 0
```

```
solid? : any/c = #t
hide-arrowhead? : any/c = #f
```

Adds a line or line-with-arrows onto *pict*, using one of the pict-finding functions (e.g., `lt-find`) to extract the source and destination of the line.

If *under?* is true, then the line and arrows are added under the existing *pict* drawing, instead of on top. If *solid?* is false, then the arrowheads are hollow instead of filled.

The *start-angle*, *end-angle*, *start-pull*, and *end-pull* arguments control the curve of the line (and the defaults produce a straight line):

- The *start-angle* and *end-angle* arguments specify the direction of curve at its start and end positions; if either is `#f`, it defaults to the angle of a straight line from the start position to end position.
- The *start-pull* and *end-pull* arguments specify a kind of momentum for the starting and ending angles; larger values preserve the angle longer.

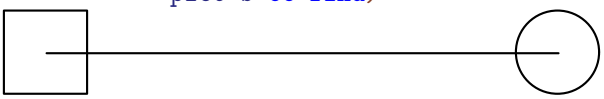
The *line-width*, *color*, *alpha*, and *style* arguments apply to the added line.

When the *hide-arrowhead?* argument is a true value, then space for an arrowhead is kept around the line, but the arrowhead itself is not drawn.

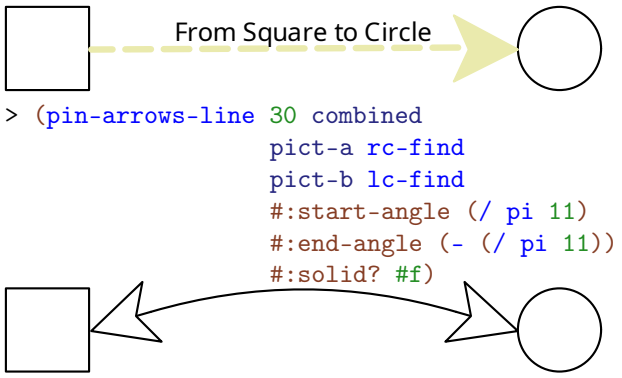
When the *label* argument is non-false, the given pict is used as a label for the line, and moved by (*x-adjust-label*, *y-adjust-label*).

Examples:

```
(define pict-a (rectangle 40 40))
(define pict-b (circle 40))
(define combined (hc-append 200 pict-a pict-b))
> (pin-line combined
            pict-a cc-find
            pict-b cc-find)
```



```
> (pin-arrow-line 30 combined
                  pict-a rc-find
                  pict-b lc-find
                  #:line-width 3
                  #:style 'long-dash
                  #:color "medium goldenrod"
                  #:label (text "From Square to Circle"))
```




```
> (pin-arrows-line 30 combined
                   pict-a rc-find
                   pict-b lc-find
                   #:start-angle (/ pi 11)
                   #:end-angle (- (/ pi 11))
                   #:solid? #f)
```

Changed in version 1.4 of package `pict-lib`: Added `#:label`, `#:x-adjust-label` and `#:y-adjust-label` arguments.

```
text-style/c : contract?
```

A contract that matches the second argument of `text`.

```
(bitmap-draft-mode) → boolean?
(bitmap-draft-mode on?) → void?
  on? : any/c
```

A parameter that determines whether `bitmap` loads/uses a bitmap.

# 3 Pict Combiners

```
(vl-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(vc-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(vr-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(ht-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(htl-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(hc-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(hbl-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
(hb-append [d] pict ...) → pict?
  d : real? = 0.0
  pict : pict-convertible?
```

Creates a new pict as a column (for `v...-append`) or row (for `h...-append`) of other picts. The optional *d* argument specifies amount of space to insert between each pair of pictures in making the column or row.

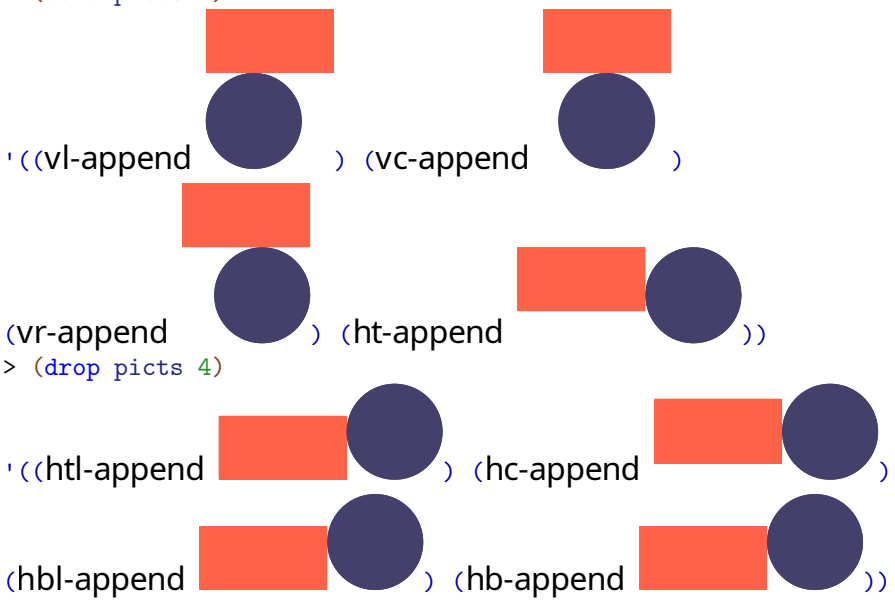
Different procedures align pictures in the orthogonal direction in different ways. For example, `vl-append` left-aligns all of the pictures.

The descent of the result corresponds to baseline that is lowest in the result among all of the picts' descent-specified baselines; similarly, the ascent of the result corresponds to the highest ascent-specified baseline. If at least one *pict* is supplied, then the last element (as reported by `pict-last`) for the result is `(or (pict-last pict) pict)` for the using last supplied *pict*.

Examples:

```
(define combiners (list vl-append vc-append vr-append
                        ht-append htl-append hc-append
                        hbl-append hb-append))
(define names (list "vl-append" "vc-append" "vr-append"
```

```
                  "ht-append" "htl-append" "hc-append"
                  "hbl-append" "hb-append"))
(define pict-a (colorize (filled-rectangle 60 30) "tomato"))
(define pict-b (colorize (disk 45) "cornflower blue"))
(define picts
  (for/list ([combiner combiners] [name names])
    (list (text name null 15)
          (combiner pict-a pict-b))))
> (take picts 4)
```




```
> (drop picts 4)
```




Changed in version 1.11 of package `pict-lib`: Removed implicit truncation of some centered coordinates to integers.

```
(lt-superimpose pict ...) → pict?
  pict : pict-convertible?
(ltl-superimpose pict ...) → pict?
  pict : pict-convertible?
(lc-superimpose pict ...) → pict?
  pict : pict-convertible?
(lbl-superimpose pict ...) → pict?
  pict : pict-convertible?
(lb-superimpose pict ...) → pict?
  pict : pict-convertible?
(ct-superimpose pict ...) → pict?
  pict : pict-convertible?
(ctl-superimpose pict ...) → pict?
  pict : pict-convertible?
```

```
(cc-superimpose pict ...) → pict?
  pict : pict-convertible?
(cbl-superimpose pict ...) → pict?
  pict : pict-convertible?
(cb-superimpose pict ...) → pict?
  pict : pict-convertible?
(rt-superimpose pict ...) → pict?
  pict : pict-convertible?
(rtl-superimpose pict ...) → pict?
  pict : pict-convertible?
(rc-superimpose pict ...) → pict?
  pict : pict-convertible?
(rbl-superimpose pict ...) → pict?
  pict : pict-convertible?
(rb-superimpose pict ...) → pict?
  pict : pict-convertible?
```

Creates a new picture by superimposing a set of pictures. The name prefixes are alignment indicators: horizontal alignment then vertical alignment.

The descent of the result corresponds to baseline that is lowest in the result among all of the picts' descent-specified baselines; similarly, the ascent of the result corresponds to the highest ascent-specified baseline. The last element (as reported by `pict-last`) for the result is the lowest, right-most among the last-element picts of the *pict* arguments, as determined by comparing the last-element bottom-right corners.

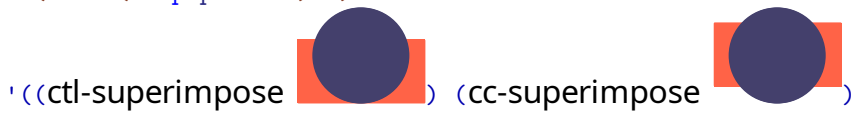
Examples:

```
(define combiners (list lt-superimpose  ltl-superimpose lc-
superimpose
                        lbl-superimpose lb-superimpose  ct-
superimpose
                        ctl-superimpose cc-superimpose  cbl-
superimpose
                        cb-superimpose  rt-superimpose  rtl-
superimpose
                        rc-superimpose  rbl-superimpose rb-
superimpose))
(define names (list "lt-superimpose"  "ltl-superimpose" "lc-
superimpose"
                    "lbl-superimpose" "lb-superimpose"  "ct-
superimpose"
                    "ctl-superimpose" "cc-superimpose"  "cbl-
superimpose"
                    "cb-superimpose"  "rt-superimpose"  "rtl-
superimpose"
```

```
                    "rc-superimpose"  "rbl-superimpose" "rb-
superimpose"))
(define pict-a (colorize (filled-rectangle 60 30) "tomato"))
(define pict-b (colorize (disk 45) "cornflower blue"))
(define picts
  (for/list ([combiner combiners] [name names])
    (list (text name null 15)
          (combiner pict-a pict-b))))
> (take picts 3)
```

```
> (take (drop picts 3) 3)
```

```
> (take (drop picts 6) 3)
```



```
> (take (drop picts 9) 3)
```

```
> (take (drop picts 12) 3)
```

(rb-superimpose ))

Changed in version 1.11 of package `pict-lib`: Removed implicit truncation of some centered coordinates to integers.

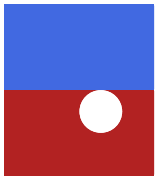
```
(pin-over base dx dy pict) → pict?
  base : pict-convertible?
  dx : real?
  dy : real?
  pict : pict-convertible?
(pin-over base find-pict find pict) → pict?
  base : pict-convertible?
  find-pict : pict-path?
  find : (pict-convertible? pict-path? . -> . (values real? real?))
  pict : pict-convertible?
```

Creates a pict with the same bounding box, ascent, and descent as *base*, but with *pict* placed on top. The *dx* and *dy* arguments specify how far right and down the second pict's corner is from the first pict's corner. Alternately, the *find-pict* and *find* arguments find a point in *base* for *find-pict*; the *find* procedure should be something like `lt-find`.

Examples:

```
> (pin-over (colorize (filled-rectangle 70 40) "chocolate")
            10 10
            (colorize (filled-rectangle 30 30) "orange"))
```

```
> (define top (colorize (filled-rectangle 70 40) "royalblue"))
> (pin-over (vc-append top (colorize (filled-
rectangle 70 40) "firebrick"))
            top
            cb-find
            (colorize (disk 20) "white"))
```



```
(pin-under base dx dy pict) → pict?
  base : pict-convertible?
```

```
  dx : real?
  dy : real?
  pict : pict-convertible?
(pin-under base find-pict find pict) → pict?
  base : pict-convertible?
  find-pict : pict-convertible?
  find : (pict-convertible? pict-path? . -> . (values real? real?))
  pict : pict-convertible?
```

Like `pin-over`, but *pict* is drawn before *base* in the resulting combination.

Examples:

```
> (define txt
    (colorize (text "P I C T S" null 25) "chocolate"))
> (define rect
    (colorize
     (filled-rectangle (pict-width txt) (* 0.3 (pict-height txt)))
     "lemonchiffon"))
> (pin-under txt
             0
             (- (/ (pict-height txt) 2)
                (/ (pict-height rect) 2))
             rect)
```

```
(table ncols
       picts
       col-aligns
       row-aligns
       col-seps
       row-seps)  → pict?
  ncols : exact-positive-integer?
  picts : (non-empty-listof pict-convertible?)
  col-aligns : (or/c (list*of (->* () #:rest (listof pict-convertible?) pict-convertible?))
                     (listof (->* () #:rest (listof pict-convertible?) pict-convertible?)))
  row-aligns : (or/c (list*of (->* () #:rest (listof pict-convertible?) pict-convertible?))
                     (listof (->* () #:rest (listof pict-convertible?) pict-convertible?)))
  col-seps : (or/c (list*of real?) (listof real?))
  row-seps : (or/c (list*of real?) (listof real?))
```

Creates a table given a list of picts. The *picts* list is a concatenation of the table's rows (which means that a Racket `list` call can be formatted to reflect the shape of the output table).

The *col-aligns*, *row-aligns*, *col-seps*, and *row-seps* arguments are "lists" specifying the row and columns alignments separation between rows and columns. For *c* columns and *r* rows, the first two should have *c* and *r* superimpose procedures, and the last two should have *c*-1 and *r*-1 numbers, respectively. The lists can be "improper" (i.e., ending in a number instead of an empty list), in which case the non-pair `cdr` is used as the value for all remaining list items that were expected. The *col-aligns* and *row-aligns* procedures are used to superimpose all of the cells in a column or row; this superimposition determines the total width or height of the column or row, and also determines the horizontal or vertical placement of each cell in the column or row.

Examples:

```
> (table 4
         (map (λ (x) (text (format "~a" x)))
              (list 1 2 3 4
                    5 6 7 8
                    9000 10 11 12))
         cc-superimpose
         cc-superimpose
         10
         10)
 1    2   3   4

 5    6   7   8

9000  10  11  12
> (table 4
         (map (λ (x) (text (format "~a" x)))
              (list 1 2 3 4
                    5 6 7 8
                    9000 10 11 12))
         rc-superimpose
         cc-superimpose
         10
         10)
   1   2   3   4

   5   6   7   8

9000  10  11  12
```

# 4 Pict Drawing Adjusters

```
(scale pict factor) → pict?
  pict : pict-convertible?
  factor : real?
(scale pict w-factor h-factor) → pict?
  pict : pict-convertible?
  w-factor : real?
  h-factor : real?
```

Scales a pict drawing, as well as its bounding box, by multiplying it current size by *factor* (if two arguments are supplied) or by multiplying the current width by *w-factor* and current height by *h-factor* (if three arguments are supplied).

The drawing is scaled by adjusting the destination dc<%>'s scale while drawing the original *pict*.

Examples:

```
> (filled-rectangle 40 40)
```

```
> (scale (filled-rectangle 40 40) 1.5)
```

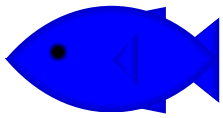
```
> (scale (filled-rectangle 40 40) 2 1.5)
```

```
(flip-x pict) → pict?
  pict : pict-convertible?
(flip-y pict) → pict?
  pict : pict-convertible?
```

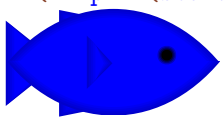
Flips a pict drawing horizontally or vertically.

Examples:

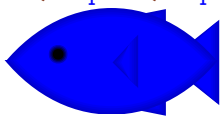
```
> (standard-fish 100 50)
```



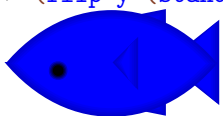
```
> (flip-x (standard-fish 100 50))
```



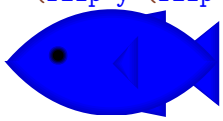
```
> (flip-x (flip-x (standard-fish 100 50)))
```



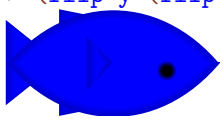
```
> (flip-y (standard-fish 100 50))
```



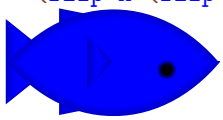
```
> (flip-y (flip-y (standard-fish 100 50)))
```



```
> (flip-y (flip-x (standard-fish 100 50)))
```



```
> (flip-x (flip-y (standard-fish 100 50)))
```



```
(scale-to-fit pict size-pict [#:mode mode]) → pict?
  pict : pict-convertible?
  size-pict : pict-convertible?
  mode : (or/c 'preserve 'inset         = 'preserve
               'preserve/max 'inset/max
               'distort)
(scale-to-fit pict width height [#:mode mode]) → pict?
  pict : pict-convertible?
  width : real?
  height : real?
  mode : (or/c 'preserve 'inset         = 'preserve
               'preserve/max 'inset/max
               'distort)
```

Scales *pict* so that it fits within the bounding box of *size-pict* (if two arguments are supplied) or into a box of size *width* by *height* (if three arguments are supplied).

If *mode* is 'preserve or 'preserve/max, the width and height are scaled by the same factor so *pict*'s aspect ratio is preserved. If *mode* is 'preserve the result's bounding box will not be larger than *width* by *height* but it may be smaller. When *mode* is 'preserve/max, the opposite is true; the bounding box will never be smaller, but might be larger.

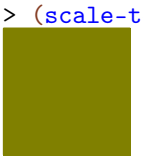
If *mode* is 'inset or 'inset/max, the aspect ratio is preserved as with 'preserve and 'preserve/max, but the resulting pict is centered on a bounding box of exactly *width* by *height*.

If *mode* is 'distort, the width and height are scaled separately.

Examples:

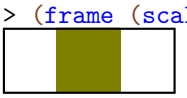
```
> (define rect (colorize (filled-rectangle 40 40) "olive"))
> rect
```

```
> (scale-to-fit rect (disk 60))
```



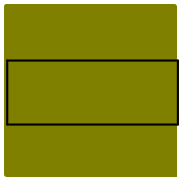
```
> (scale-to-fit rect 80 30 #:mode 'preserve)
```

```
> (frame (scale-to-fit rect 80 30 #:mode 'inset))
```



```
> (scale-to-fit rect 80 30 #:mode 'preserve/max)
```

```
> (cc-superimpose (blank 100 100)
                  (frame (scale-to-fit rect 80 30 #:mode 'inset/max)))
```



```
> (scale-to-fit rect 80 30 #:mode 'distort)
```

Changed in version 1.4 of package `pict-lib`: Added `#:mode` argument.

```
(rotate pict theta) → pict?
  pict : pict-convertible?
  theta : real?
```

Rotates a pict's drawing by *theta* radians counter-clockwise.

The bounding box of the resulting pict is the box encloses the rotated corners of *pict* (which inflates the area of the bounding box, unless *theta* is a multiple of half of `pi`). The ascent and descent lines of the result's bounding box are the horizontal lines that bisect the rotated original lines; if the ascent line drops below the descent line, the two lines are flipped.

Examples:

```
> (rotate (colorize (filled-rectangle 30 30)
                    "chartreuse")
          (/ pi 3))
```

```
> (explain (rotate tt 0))
```

```
> (explain (rotate tt (/ pi 2)))
```

```
> (explain (rotate tt pi))
```

```
(shear pict x-shear y-shear) → pict?
  pict : pict-convertible?
  x-shear : number?
  y-shear : number?
```

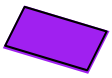
Shear's the *pict* by a factor of *x-shear* along the x-axis and by a factor of *y-shear* along the y-axis. The bounding box is inflated to contain the result. The result's ascent and descent are the same as *pict*'s.

Examples:

```
> (define sqr
    (colorize (filled-rectangle 40 20 #:draw-
border? #t #:color "purple")
              "black"))
> (shear sqr 0 0.3)
```

```
> (shear sqr -0.5 0)
```

```
> (shear sqr -0.5 0.3)
```



Added in version 1.8 of package `pict-lib`.

```
(translate pict dx dy [#:extend-bb? bb?]) → pict?
```

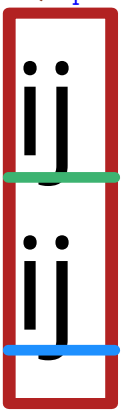
```
pict : pict-convertible?
dx : real?
dy : real?
bb? : any/c = #f
```

Translate the *pict* by *dx* and *dy*. If *bb?* is not #f then the bounding box is extended to ensure the image fits.

Examples:

```
> (explain (translate tt -3 -3))
```

```
> (explain (translate tt -3 -3 #:extend-bb? #t))
```



```
> (explain (translate tt 3 3))
```

```
> (explain (translate tt 3 3 #:extend-bb? #t))
```

Added in version 1.10 of package `pict-lib`.

```
(ghost pict) → pict?
  pict : pict-convertible?
```

Creates a container picture that doesn't draw the child picture, but uses the child's size.

Example:

```
> (frame (hc-append (ghost (filled-rectangle 30 30))
                    (colorize (disk 30) "turquoise")))
```

```
(linewidth w pict) → pict?
  w : (or/c real? #f)
  pict : pict-convertible?
```

Selects a specific pen width for drawing, which applies to pen drawing for *pict* that does not already use a specific pen width. A `#f` value for *w* makes the pen transparent (in contrast to a zero value, which means “as thin as possible for the target device”).

Examples:

```
> (linewidth 3 (hline 40 1))
> (linewidth 5 (hline 40 1))
```

```
(linestyle style pict) → pict?
  style : (or/c 'transparent 'solid 'xor 'hilite
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
  pict : pict-convertible?
```

Selects a specific pen style for drawing, which applies to pen drawing for *pict* that does not already use a specific pen style.

Examples:

```
> (define styles
    '(transparent solid xor hilite dot long-dash short-dash
      dot-dash xor-dot xor-long-dash xor-short-dash xor-dot-dash))
> (apply ht-append
         10
         (for/list ([style (in-list styles)])
           (vc-append 5
             (text (symbol->string style))
             (linewidth 3 (linestyle style (hline 40 1))))))
```

transparent solid xor hilite dot long-dash short-dash dot-dash xor-dot xor-long-dash

```
(colorize pict color) → pict?
  pict : pict-convertible?
  color : (or/c string? (is-a?/c color%)
                (list/c byte? byte? byte?))
```

Selects a specific color drawing, which applies to drawing in *pict* that does not already use a specific color. The `black-and-white` parameter causes all non-white colors to be converted to black.

Examples:

```
> (colorize (disk 40) "lavender")
```



```
> (colorize (filled-rectangle 40 40)
            (list 255 153 85))
```

```
> (colorize (arrow 40 0)
            (make-color 170 180 120))
```

```
(cellophane pict
            opacity
            [#:composite? composite?]) → pict?
  pict : pict-convertible?
  opacity : (real-in 0 1)
  composite? : any/c = #t
```

Makes the given *pict* semi-transparent, where an opacity of 0 is fully transparent, and an opacity of 1 is fully opaque.

If *composite?* is a true value, then the opacity adjustment is applied to *pict* as a whole, instead of to individual drawing operations within *pict*. Rendering *pict* with compositing may take longer than with *composite?* as #false.

Examples:

```
> (cc-superimpose (filled-rectangle 70 45 #:color "darkcyan")
                  (cellophane (disk 40) 0.2))
```

```
> (cc-superimpose (filled-rectangle 70 45 #:color "darkcyan")
                  (cellophane (disk 40) 0.8))
```

Changed in version 1.15 of package pict-lib: Added #:composite? and enabled compositing by default.

```
(clip pict) → pict
  pict : pict-convertible?
```

Clips a pict's drawing to its bounding box.

Examples:

```
> (define shape
    (inset (colorize (filled-rectangle 40 40) "thistle") -10))
> shape
```

```
> (clip shape)
```



```
(inset/clip pict amt) → pict?
  pict : pict-convertible?
  amt : real?
(inset/clip pict h-amt v-amt) → pict?
  pict : pict-convertible?
  h-amt : real?
  v-amt : real?
(inset/clip pict l-amt t-amt r-amt b-amt) → pict?
  pict : pict-convertible?
  l-amt : real?
  t-amt : real?
  r-amt : real?
  b-amt : real?
```

Insets and clips the pict's drawing to its bounding box. Usually, the inset amounts are negative.

Examples:

```
> (filled-rectangle 40 40 #:color "forestgreen")
```

```
> (inset/clip (filled-rectangle 40 40 #:color "forestgreen") -10)
```

```
> (inset/clip (filled-rectangle 40 40 #:color "forestgreen")
              -10 -5)
```

```
> (inset/clip (filled-rectangle 40 40 #:color "forestgreen")
              -2 -4 -8 -16)
```

```
(scale/improve-new-text pict-expr scale-expr)
(scale/improve-new-text pict-expr x-scale-expr y-scale-expr)
```

Like the scale procedure, but also sets current-expected-text-scale while evaluating *pict-expr*.

Examples:

```
> (text "Hello World" null 25)
```

Hello World

```
> (scale/improve-new-text (text "Hello World" null 25) 2)
```

Hello World

```
> (scale (text "Hello World" null 25) 2)
```

Hello World

```
(black-and-white) → boolean?
(black-and-white on?) → void?
  on? : any/c
```

A parameter that determines whether colorize uses color or black-and-white colors.

Examples:

```
> (colorize (disk 40) "seagreen")
```

```
> (parameterize ([black-and-white #t])
    (colorize (disk 40) "seagreen"))
```

```
(freeze pict
        [#:scale scale-amount
         #:inset inset-amount]) → pict?
```

```
pict : pict-convertible?
scale-amount : real? = 1
inset-amount : (or/c real?                                  = 0
                     (list/c real?)
                     (list/c real? real?)
                     (list/c real? real? real? real?))
```

Creates a bitmap with the same size as *pict*, draws *pict* into the bitmap, and then returns a pict that draws with the bitmap.

This has the effect of speeding up subsequent drawing of the pict and also of cropping it to its bounding box. Any sub-picts of *pict* remain intact within the new pict.

The *scale-amount* argument controls the size of the bitmap used to render the pict. For example, if *scale-amount* is 2, then a bitmap that's twice as wide and twice as tall as the pict is used to draw into. Regardless of the value of *scale-amount*, the result pict's size is that same as *pict*'s size.

The *inset-amount* argument allows `freeze` to save more of the area that the pict draws in, which is useful when a pict draws outside of its bounding box. The values are used as the arguments to `inset`. As with the *scale-amount* argument, the size of the resulting pict is the same as the size of the input pict; this argument controls the size of the bitmap.

Examples:

```
> (define txt
    (colorize (text "Freeze!" null 25) "deepskyblue"))
> (scale txt 2.5)
```

```
> (scale (freeze txt) 2.5)
```

Changed in version 1.13 of package `pict-lib`: Added the `scale` and `inset` arguments.

# 5 Bounding Box Adjusters

```
(inset pict amt) → pict?
  pict : pict-convertible?
  amt : real?
(inset pict h-amt v-amt) → pict?
  pict : pict-convertible?
  h-amt : real?
  v-amt : real?
(inset pict l-amt t-amt r-amt b-amt) → pict?
  pict : pict-convertible?
  l-amt : real?
  t-amt : real?
  r-amt : real?
  b-amt : real?
```

Extends *pict*'s bounding box by adding the given amounts to the corresponding sides; ascent and descent are extended, too.

Examples:

```
> (pict-width (disk 40))
40
> (pict-width (inset (disk 40) -10))
20
```

```
(clip-descent pict) → pict?
  pict : pict-convertible?
```

Truncates *pict*'s bounding box by removing the descent part.

Examples:

```
> (explain (text "gjy" null 12))
```

```
> (explain (clip-descent (text "gjy" null 12)))
```

```
(clip-ascent pict) → pict?
  pict : pict-convertible?
```

Truncates *pict*'s bounding box by removing the ascent part.

Examples:

```
> (explain (text "gjy" null 12))
```

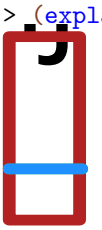
```
> (explain (clip-ascent (text "gjy" null 12)))
```

```
(lift-bottom-relative-to-baseline  p
                                   baseline
                                  [#:extend-bb? bb?]) → pict?
  p : pict-convertible?
  baseline : real?
  bb? : any/c = #f
```

Lift the image in *p* such that the bottom of the picture is *baseline* above its baseline. If *bb?* is #f then the bounding box is unchanged, otherwise it is adjusted to fit the resulting pict.

Examples:

```
> (define shift 10)
> (explain (lift-bottom-relative-to-baseline t shift))
```

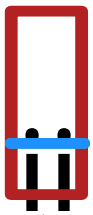


```
> (explain (lift-bottom-relative-to-baseline t shift #:extend-
bb? #t))
```

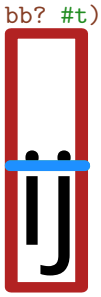
```
> (explain (lift-bottom-relative-to-baseline tt shift))
```

```
> (explain (lift-bottom-relative-to-baseline tt shift #:extend-
bb? #t))
```

```
> (explain (lift-bottom-relative-to-baseline t (- shift)))
```



```
> (explain (lift-bottom-relative-to-baseline t (- shift) #:extend-
bb? #t))
```



```
> (explain (lift-bottom-relative-to-baseline tt (- shift)))
```

```
> (explain (lift-bottom-relative-to-baseline tt (- shift) #:extend-
bb? #t))
```

Added in version 1.10 of package `pict-lib`.

```
(drop-top-relative-to-ascent p
                             ascent
                             [#:extend-bb? bb?]) → pict?
  p : pict-convertible?
  ascent : real?
  bb? : any/c = #f
```

Lift the image in *p* such that the top of the picture is *ascent* below its ascent. If *bb?* is `#f` then the bounding box is unchanged, otherwise it is adjusted to fit the resulting pict.

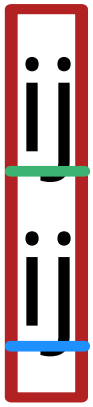
Examples:

```
> (define shift 10)
> (explain (drop-top-relative-to-ascent t shift))
```

```
> (explain (drop-top-relative-to-ascent t shift #:extend-bb? #t))
```

```
> (explain (drop-top-relative-to-ascent tt shift))
```

```
> (explain (drop-top-relative-to-ascent tt shift #:extend-bb? #t))
```



```
> (explain (drop-top-relative-to-ascent t (- shift)))
```

```
> (explain (drop-top-relative-to-ascent t (- shift) #:extend-
bb? #t))
```

```
> (explain (drop-top-relative-to-ascent tt (- shift)))
```

```
> (explain (drop-top-relative-to-ascent tt (- shift) #:extend-
bb? #t))
```

Added in version 1.10 of package `pict-lib`.

```
(lift-above-baseline pict amt) → pict?
  pict : pict-convertible?
  amt : real?
```

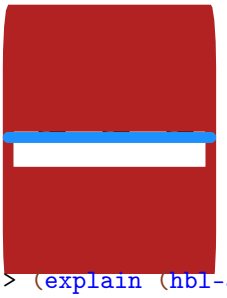
For backwards compatibility. Use `lift-bottom-relative-to-baseline` instead.

Lifts *pict* relative to its baseline, extending the bounding box height if necessary.

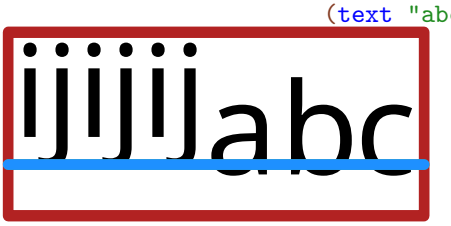
Examples:

```
> (explain (hbl-append (text "ijijij" null 12)
                       (text "abc" null 12)))
```

```
> (inset (explain (lift-above-baseline (text "ijijij" null 12) 20)) 0 100 0 50)
```



```
> (explain (hbl-append (lift-above-baseline (text "ijijij" null 12) 20)
                       (text "abc" null 12)))
```



```
(drop-below-ascent pict amt) → pict?
  pict : pict-convertible?
  amt : real?
```

For backwards compatibility. Use `drop-top-relative-to-ascent` instead.

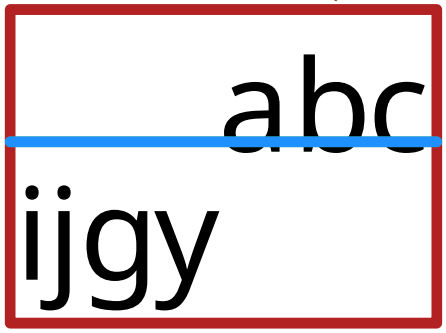
Drops *pict* relative to its ascent line, extending the bounding box height if necessary.

Examples:

```
> (define txt (text "ijgy" null 12))
> (explain (hbl-append txt (text "abc" null 12)))
```

```
> (explain (drop-below-ascent txt 12))
```

```
> (explain (hbl-append (drop-below-ascent txt 12)
                       (text "abc" null 12)))
```



```
(baseless pict) → pict?
  pict : pict-convertible?
```

Makes the descent 0 and the ascent the same as the height.

Examples:

```
> (explain (hbl-append (text "gjy" null 12)
                       (text "abc" null 12)))
```

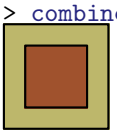
```
> (explain (hbl-append (baseless (text "gjy" null 12))
                       (text "abc" null 12)))
```

(refocus *pict* *sub-pict*) → pict?
  *pict* : pict-convertible?
  *sub-pict* : pict-convertible?

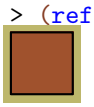
Assuming that *sub-pict* can be found within *pict*, shifts the overall bounding box to that of *sub-pict* (but preserving all the drawing of *pict*). The last element, as reported by pict-last is also set to (or (pict-last *sub-pict*) *sub-pict*).

Examples:

```
> (define p1 (filled-rectangle 50 50 #:color "darkkhaki"))
> (define p2 (filled-rectangle 30 30 #:color "sienna"))
> (define combined (cc-superimpose p1 p2))
> combined
```



```
> (refocus combined p2)
```

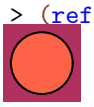


(panorama *pict*) → pict?
  *pict* : pict-convertible?

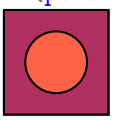
Shifts the given pict's bounding box to enclose the bounding boxes of all sub-picts (even laundered picts).

Examples:

```
> (define p1 (filled-rectangle 50 50 #:color "maroon"))
> (define p2 (disk 30 #:color "tomato"))
> (define combined (cc-superimpose p1 p2))
> (refocus combined p2)
```



```
> (panorama (refocus combined p2))
```



```
(use-last pict sub-pict) → pict?
  pict : pict-convertible?
  sub-pict : pict-path?
```

Returns a pict like *pict*, but with the last element (as reported by pict-last) set to *sub-pict*. The *sub-pict* must exist as a sub-pict (or path of sub-picts) within *pict*.

```
(use-last* pict sub-pict) → pict?
  pict : pict-convertible?
  sub-pict : pict-convertible?
```

Propagates the last element of *sub-pict* to *pict*.

That is, use-last* is like use-last, but the last element of *sub-pict* is used as the new last element for *pict*, instead of *sub-pict* itself—unless (pict-last *sub-pict*) is #f, in which case *sub-pict* is used as the last element of *pict*.

# 6 Pict Finders

```
(lt-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(ltl-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(lc-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(lbl-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(lb-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(ct-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(ctl-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(cc-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(cbl-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(cb-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(rt-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
```

```
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(rtl-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(rc-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(rbl-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
(rb-find pict find [#:nth nth]) → real? real?
  pict : pict-convertible?
  find : pict-path?
  nth : (or/c exact-nonnegative-integer? 'unique) = 0
```

Locates a pict designated by *find* within *pict*. If *find* is a pict, then the *pict* must have been created as some combination involving *find*.

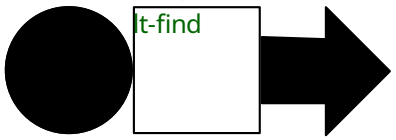
If *find* is a list, then the first element of *find* must be within *pict*, the second element of *find* must be within the first element, and so on. In general, the $i$+1st element must be within the $i$th element, so the list can provide a path to some specific pict, in case there is more than one occurrence of the pict that's being searched for within *pict*.

If the last pict in *find* exists multiple times in the enclosing pict (either *pict* or the next-to-last element of *find*), then the location for the first found occurrence is returned by default. If *nth* is a larger number, then then first *nth* found locations are skipped. If *nth* is `'unique`, then all instances are found, and an error is reported if multiple instances are found at different locations.

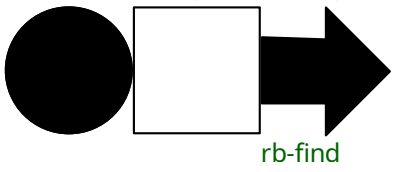
Examples:

```
> (define p1 (disk 60))
> (define p2 (rectangle 60 60))
> (define p3 (hc-append p1 p2))
> (define p4 (hc-append p3 (arrow 60 0)))
> (lt-find p4 p1)
0
0
> (cb-find p4 p2)
90
60
> (rb-find p3 p1)
60
```

```
60
> (lt-find p4 (list p1))
0
0
> (lt-find p4 (list p2 p1))
lt-find: sub-pict not found
  sub-pict: #<pict>
  in: #<pict>
> (lt-find p4 (list p2))
60
0
> (lt-find p4 (list p3 p2))
60
0
> (pin-over p4 p2 lt-find
            (colorize (text "lt-find") "darkgreen"))
```




```
> (panorama
   (pin-over p4 p2 rb-find
             (colorize (text "rb-find") "darkgreen")))
```




Changed in version 1.11 of package `pict-lib`: Removed implicit truncation of some centered coordinates to integers.
Changed in version 1.16: Added the *nth* argument.

```
(pict-path? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a `pict-convertible?` or a non-empty list of `pict-convertible?`s.

Examples:

```
> (pict-path? null)
#f
> (pict-path? (disk 30))
#t
> (pict-path? (list (disk 30) (rectangle 10 10)))
```

```
#t
```

```
(launder pict) → pict?
  pict : pict-convertible?
```

Creates a pict that has the same drawing and bounding box of *pict*, but which hides all of its sub-picts so that they cannot be found with functions like `lt-find`. If *pict* has a last-line pict, then the laundered pict has a fresh last-line pict with the same shape and location.

Examples:

```
> (define p1 (disk 60))
> (define p2 (rectangle 60 60))
> (define p3 (hc-append p1 p2))
> (lt-find (launder p4) p3)
lt-find: sub-pict not found
  sub-pict: #<pict>
  in: #<pict>
```

# 7 More Pict Constructors

## 7.1 Dingbats

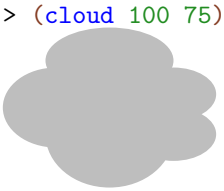
```
(cloud w h [color #:style style]) → pict?
  w : real?
  h : real?
  color : (or/c string? (is-a?/c color%)) = "gray"
  style : (listof (or/c 'square          = null
                        'nw 'ne 'sw 'se
                         'wide))
```

Creates a fluffy cloud.

Supplying `'nw` (northwest), `'ne` (northeast), `'sw` (southwest), or `'se` (southeast) in *style* extends the cloud to better cover the corner within its bounding box in the corresponding direction. The `'square` style is a shorthand for all of those combined. The `'wide` style increases the number of bumps along the cloud's top and bottom edges.

Examples:

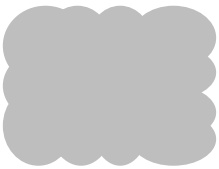
```
> (cloud 100 75)
```



```
> (cloud 100 75 "lavenderblush")
```



```
> (cloud 100 75 #:style '(square wide))
```



Changed in version 1.6 of package `pict-lib`: Added *style*.

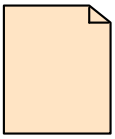
```
(file-icon w h color [shaded?]) → pict?
  w : real?
```

```
  h : real?
  color : (or/c string? (is-a?/c color%) any/c)
  shaded? : any/c = #f
```

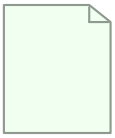
Creates a Mac-like file icon, optionally shaded. If *color* is not a string or color% object, it is treated as a boolean, in which case true means "gray" and false means "white".

Examples:

```
> (file-icon 50 60 "bisque")
```



```
> (file-icon 50 60 "honeydew" #t)
```



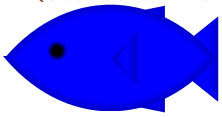
```
(standard-fish  w
                h
               [#:direction direction
                #:color color
                #:eye-color eye-color
                #:open-mouth open-mouth]) → pict?
  w : real?
  h : real?
  direction : (or/c 'left 'right) = 'left
  color : (or/c string? (is-a?/c color%)) = "blue"
  eye-color : (or/c string? (is-a?/c color%) #f) = "black"
  open-mouth : (or/c boolean? real?) = #f
```

Creates a fish swimming either 'left or 'right. If *eye-color* is #f, no eye is drawn.

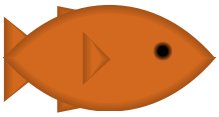
The *open-mouth* argument can be either #f (mouth closed), #t (mouth fully open), or a number: 0.0 is closed, 1.0 is fully open, and numbers in between are partially open.

Examples:

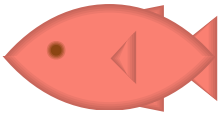
```
> (standard-fish 100 50)
```



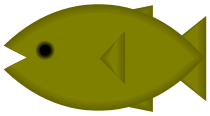
```
> (standard-fish 100 50 #:direction 'right #:color "chocolate")
```



```
> (standard-fish 100 50 #:eye-color "saddlebrown" #:color "salmon")
```



```
> (standard-fish 100 50 #:open-mouth #t #:color "olive")
```



```
(jack-o-lantern  size
                 [pumpkin-color
                  face-color
                  stem-color])   → pict?
  size : real?
  pumpkin-color : (or/c string? (is-a?/c color%)) = "orange"
  face-color : (or/c string? (is-a?/c color%)) = "black"
  stem-color : (or/c string? (is-a?/c color%)) = "brown"
```

Creates a jack-o-lantern; use the same pumpkin and face color to get a plain pumpkin. The *size* determines the width.

Examples:

```
> (jack-o-lantern 100)
```

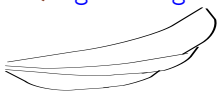
```
> (jack-o-lantern 100 "cadet blue" "khaki" "salmon")
```

```
(angel-wing w h left?) → pict?
  w : real?
  h : real?
  left? : any/c
```

Creates an angel wing, left or right, or any size. The color and pen width for drawing the wing outline is the current one.

Examples:

```
> (angel-wing 100 40 #f)
```



```
> (angel-wing 100 40 #t)
```

```
(desktop-machine scale [style]) → pict?
  scale : real?
  style : (listof symbol?) = null
```

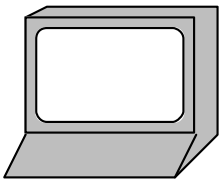
Produces a picture of ancient desktop computer. The *scale* argument scales the size relative to the base size of 120 by 115.

The *style* can include any of the following:

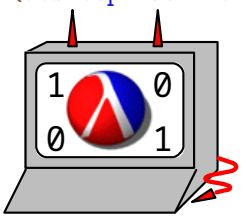
- 'plt — include a Racket logo on the machine's screen
- 'binary — put 1s and 0s on the machine's screen
- 'devil — like 'binary, and also give the machine horns and a tail

Examples:

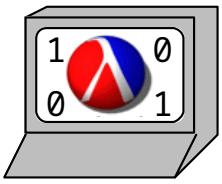
```
> (desktop-machine 1)
```



```
> (desktop-machine 1 '(devil plt))
```




```
> (desktop-machine 1 '(plt binary))
```




```
(thermometer [#:height-% height-%
              #:color-% color-%
              #:ticks ticks
              #:start-color start-color
              #:end-color end-color
              #:top-circle-diameter top-circle-diameter
              #:bottom-circle-diameter bottom-circle-diameter
              #:stem-height stem-height
              #:mercury-inset mercury-inset])
 → pict?
  height-% : (between/c 0 1) = 1
  color-% : (between/c 0 1) = height-%
  ticks : exact-nonnegative-integer? = 4
  start-color : (or/c string? (is-a?/c color%)) = "lightblue"
  end-color : (or/c string? (is-a?/c color%)) = "lightcoral"
  top-circle-diameter : (>/c 0) = 40
  bottom-circle-diameter : (>/c 0) = 80
  stem-height : (>/c 0) = 180
  mercury-inset : (>/c 0) = 8
```

Produces a thermometer that consists of a semi-circle on top of a rectangle on top of a circle. The sizes of the three components are controlled via the *top-circle-diameter*, *stem-height*, and *bottom-circle-diameter* arguments.

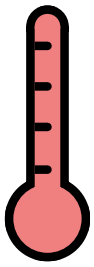
The mercury is drawn the same way, but by creating the three components inset from the versions that draw the boundary of the thermometer. This inset is controlled by the *mercury-inset* argument.

The height of the mercury in the thermometer is controlled by the *height-%* argument. Its color is interpolated between the *start-color* and *end-color*, as determined by the *color-%* argument.

Finally, some number of ticks are drawn, based on the *ticks* argument.

Example:

```
> (thermometer #:stem-height 90
               #:bottom-circle-diameter 40
               #:top-circle-diameter 20
               #:mercury-inset 4)
```



```
(standard-cat width
              height
              #:left-ear-extent left-ear-extent
              #:left-ear-arc left-ear-arc
              #:left-ear-angle left-ear-angle
              #:right-ear-extent right-ear-extent
              #:right-ear-arc right-ear-arc
              #:right-ear-angle right-ear-angle
              #:fur-color fur-color
              #:fur-border-color fur-border-color
              #:lip-color lip-color
              #:lip-border-color lip-border-color
              #:lip-border-width lip-border-width
              #:eye-color eye-color
              #:nose-color nose-color
              #:nose nose
              #:happy? happy?
              #:eyes eyes
              #:left-eye left-eye
              #:right-eye right-eye
              #:whisker-length whisker-length
              #:whisker-droop whisker-droop
              #:whisker-width whisker-width
              #:whisker-color whisker-color
              #:whisker-inset? whisker-inset?
              #:border-width border-width)        → pict?
```

```
width : positive?
height : positive?
left-ear-extent : (>=/c 0)
left-ear-arc : (real-in 0 (* 2 pi))
left-ear-angle : (real-in 0 (* 2 pi))
right-ear-extent : (>=/c 0)
right-ear-arc : (real-in 0 (* 2 pi))
right-ear-angle : (real-in 0 (* 2 pi))
fur-color : (or/c #f string? (is-a?/c color%))
fur-border-color : (or/c #f string? (is-a?/c color%))
lip-color : (or/c #f string? (is-a?/c color%))
lip-border-color : (or/c string? (is-a?/c color%))
lip-border-width : (or/c #f (real-in 0 255))
eye-color : (or/c #f string? (is-a?/c color%))
nose-color : (or/c #f string? (is-a?/c color%))
nose : pict?
happy? : any/c
eyes : (or/c #f pict?)
left-eye : (if eyes (or/c #f pict?) pict?)
right-eye : (if eyes (or/c #f pict?) pict?)
whisker-length : positive?
whisker-droop : real?
whisker-width : (or/c #f (real-in 0 255))
whisker-color : (or/c string? (is-a?/c color%))
whisker-inset? : any/c
border-width : (or/c #f (real-in 0 255))
```

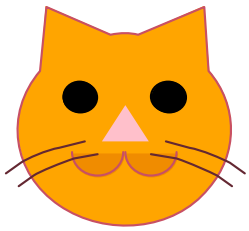
Creates a cat face with customizable whiskers, eyes, ears, and mouth.

The *whisker-width* argument can be either #f (no whiskers) or a real between 0 and 255 to specify pen width. Similarly for *border-width*, there can be no border around the cat face or a custom-width border.

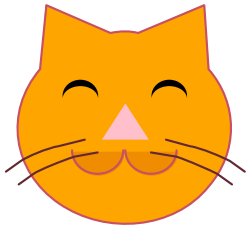
Each cat ear can have a custom extent (length), an arc length (width), and angle (position on head). There are constraints on the ear arc lengths and angles to prevent overlap. The default ear angle is dependent on the ear arc length to make the default positioning sensible. The default ear extent is 1/4 the height of the cat face.

Examples:

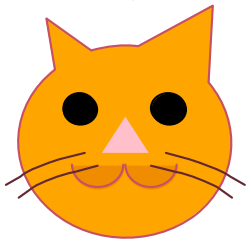
```
> (standard-cat 100 90)
```



```
> (standard-cat 100 90 #:happy? #t)
```



```
> (standard-cat 100 90 #:left-ear-arc (* pi 1/8) #:right-ear-
extent 30)
```



Added in version 1.10 of package `pict-lib`.

## 7.2 Balloon Annotations

```
(require pict/balloon)
```
package: `pict-lib`

The `pict/balloon` library provides functions for creating and placing cartoon-speech balloons.

```
(wrap-balloon pict
              spike
              dx
              dy
             [color
              corner-radius]) → balloon?
  pict : pict?
  spike : (or/c 'n 's 'e 'w 'ne 'se 'sw 'nw)
  dx : real?
  dy : real?
```

```
  color : (or/c string? (is-a?/c color%))
        = (current-balloon-color)
  corner-radius : (and/c real? (not/c negative?)) = 32
```

Superimposes *pict* on top of a balloon that wraps it.

The *spike* argument indicates the corner from which a spike protrudes from the balloon (i.e., the spike that points to whatever the balloon is about). For example, `'n` means "north,", which is a spike in the top middle of the balloon.

The *dx* and *dy* arguments specify how far the spike should protrude. For a `'w` spike, *dx* should be negative, etc.

The *color* argument is the background color for the balloon.

The *corner-radius* argument determines the radius of the circle used to round the balloon's corners. As usual, if it is less than `1`, then it acts as a ratio of the balloon's width or height.

The result is a balloon, not a pict. The `balloon-pict` function extracts a pict whose bounding box does not include the spike, but includes the rest of the image, and the `balloon-point-x` and `balloon-point-y` functions extract the location of the spike point. More typically, the `pin-balloon` function is used to add a balloon to a pict.

```
(pip-wrap-balloon  pict
                   spike
                   dx
                   dy
                  [color
                   corner-radius]) → pict?
  pict : pict?
  spike : (or/c 'n 's 'e 'w 'ne 'se 'sw 'nw)
  dx : real?
  dy : real?
  color : (or/c string? (is-a?/c color%))
        = (current-balloon-color)
  corner-radius : (and/c real? (not/c negative?)) = 32
```

Like `wrap-balloon`, but produces a zero-sized pict suitable for use with `pin-over`.

```
(pin-balloon balloon base x y) → pict?
  balloon : balloon?
  base : pict?
  x : real?
  y : real?
(pin-balloon balloon base at-pict find) → pict?
```

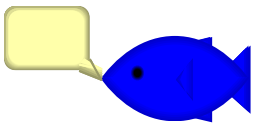
```
balloon : balloon?
base : pict?
at-pict : pict-path?
find : (pict? pict-path? . -> . (values real? real?))
```

Superimposes the pict in *balloon* onto *base* to produce a new pict. The balloon is positioned so that its spike points to the location specified by either *x* and *y* (numbers) or at the position determined by combining *base* and *at-pict* with *find*. The *find* function uses its arguments like `lt-find`.

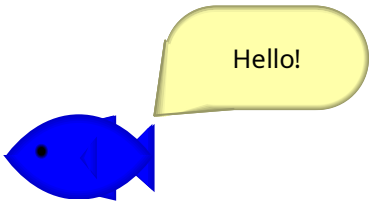
The resulting pict has the same bounding box, descent, and ascent as *base*, even if the balloon extends beyond the bounding box.

Examples:

```
> (define a-pict (standard-fish 70 40))
> (pin-balloon (balloon 40 30 5 'se 5 5)
               (cc-superimpose (blank 300 150) a-pict)
               a-pict
               lc-find)
```



```
> (pin-balloon (wrap-balloon (text "Hello!") 'sw -5 3)
               (cc-superimpose (blank 300 150) a-pict)
               a-pict
               rt-find)
```



```
(balloon w h corner-radius spike dx dy [color]) → balloon?
  w : real?
  h : real?
  corner-radius : (and/c real? (not/c negative?))
  spike : (or/c 'n 's 'e 'w 'ne 'se 'sw 'nw)
  dx : real?
  dy : real?
  color : (or/c string? (is-a?/c color%))
        = (current-balloon-color)
```

Creates a balloon, much like `wrap-balloon` except that the balloon's width is `w` and its height is `h`.

```
(balloon? v) → boolean?
  v : any/c
(make-balloon pict x y) → balloon?
  pict : pict?
  x : real?
  y : real?
(balloon-pict balloon) → pict?
  balloon : balloon?
(balloon-point-x balloon) → real?
  balloon : balloon?
(balloon-point-y balloon) → real?
  balloon : balloon?
```

A balloon encapsulates a pict and the position of the balloon's spike relative to the balloon's top-left corner.

```
balloon-color : (or/c string? (is-a?/c color%))
```

The default background color for a balloon.

```
(current-balloon-color) → (or/c string? (is-a?/c color%))
(current-balloon-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

Determines the background color for a balloon as created by functions like `wrap-balloon`.

Added in version 1.9 of package `pict-lib`.

```
(balloon-enable-3d) → boolean?
(balloon-enable-3d on?) → void?
  on? : any/c
```

A parameter that determines whether balloons are drawn with 3-D shading. This parameter affects balloons at drawing time, not at construction time.

## 7.3 Face

```
(require pict/face)
```
package: pict-lib

The pict/face library provides functions for a kind of Mr. Potatohead-style face library.

```
default-face-color : (or/c string (is-a?/c color%))
```

Orange.

```
(face mood [color]) → pict?
  mood : symbol?
  color : (or/c string (is-a?/c color%)) = default-face-color
```

Returns a pict for a pre-configured face with the given base color. The built-in configurations, selected by mood-symbol, are as follows:

'unhappy — (face* 'none 'plain #t default-face-color 6)

'sortof-unhappy — (face* 'worried 'grimace #t default-face-color 6)

'sortof-happy — (face* 'worried 'medium #f default-face-color 6)

'happy — (face* 'none 'plain #f default-face-color 6)

'happier — (face* 'none 'large #f default-face-color 3)

```
'embarrassed — (face* 'worried 'medium #f default-face-color 3)
```

```
'badly-embarrassed — (face* 'worried 'medium #t default-face-color 3)
```

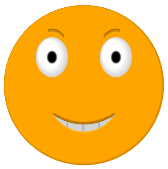
```
'unhappier — (face* 'normal 'large #t default-face-color 3)
```

```
'happiest — (face* 'normal 'huge #f default-face-color 0 -3)
```

```
'unhappiest — (face* 'normal 'huge #t default-face-color 0 -3)
```

```
'mad — (face* 'angry 'grimace #t default-face-color 0)
```

```
'mean — (face* 'angry 'narrow #f default-face-color 0)
```



```
'surprised — (face* 'worried 'oh #t default-face-color -4 -3 2)
```

```
(face* eyebrow-kind
       mouth-kind
       frown?
       color
       eye-inset
       eyebrow-dy
       pupil-dx
       pupil-dy
      [#:eyebrow-shading? eyebrow-on?
       #:mouth-shading? mouth-on?
       #:eye-shading? eye-on?
       #:tongue-shading? tongue-on?
       #:face-background-shading? face-bg-on?
       #:teeth? teeth-on?])                    → pict?
  eyebrow-kind : (or/c 'none 'normal 'worried 'angry)
  mouth-kind : (or/c 'plain 'smaller 'narrow 'medium 'large
                     'huge 'grimace 'oh 'tongue)
  frown? : any/c
  color : (or/c string (is-a?/c color%))
  eye-inset : real?
  eyebrow-dy : real?
  pupil-dx : real?
  pupil-dy : real?
  eyebrow-on? : any/c = #t
  mouth-on? : any/c = #t
  eye-on? : any/c = #t
  tongue-on? : any/c = #t
  face-bg-on? : any/c = #t
  teeth-on? : any/c = #t
```

Returns a pict for a face:

- *eyebrow-kind* determines the eyebrow shape.
- *mouth-kind* determines the mouth shape, combined with *frown?*.
- *frown?* determines whether the mouth is up or down.
- *color* determines the face color.
- *eye-inset* adjusts the eye size; recommend values are between 0 and 10.
- *eyebrow-dy* adjusts the eyebrows; recommend values: between -5 and 5.
- *pupil-dx* adjusts the pupil; recommend values are between -10 and 10.
- *pupil-dy* adjusts the pupil; recommend values are between -15 and 15.

The #:eyebrow-shading? through #:face-background-shading? arguments control whether a shading is used for on a particular feature in the face (shading tends to look worse than just anti-aliasing when the face is small). The #:teeth? argument controls the visibility of the teeth for some mouth shapes.

## 7.4 Flash

```
(require pict/flash)
```
package: pict-lib

```
(filled-flash  width
               height
              [n-points
               spike-fraction
               rotation])      → pict?
  width : real?
  height : real?
  n-points : exact-positive-integer? = 10
  spike-fraction : (real-in 0 1) = 0.25
  rotation : real? = 0
```

Returns a pict for a "flash": a spiky oval, like the yellow background that goes behind a "new!" logo on web pages or a box of cereal.

The *height* and *width* arguments determine the size of the oval in which the flash is drawn, prior to rotation. The actual height and width may be smaller if points is not a multiple of 4, and the actual height and width will be different if the flash is rotated.

The *n-points* argument determines the number of points on the flash.

The *spike-fraction* argument determines how big the flash spikes are compared to the bounding oval.

The *rotation* argument specifies an angle in radians for counter-clockwise rotation.

The flash is drawn in the default color.

Examples:

```
> (filled-flash 100 50)
```

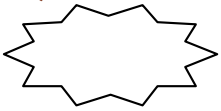
```
> (filled-flash 100 50 8 0.25 (/ pi 2))
```

```
(outline-flash  width
                height
               [n-points
                spike-fraction
                rotation])      → pict?
  width : real?
  height : real?
  n-points : exact-positive-integer? = 10
  spike-fraction : (real-in 0 1) = 0.25
  rotation : real? = 0
```

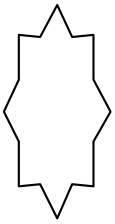
Like `filled-flash`, but drawing only the outline.

Examples:

```
> (outline-flash 100 50)
```



```
> (outline-flash 100 50 8 0.25 (/ pi 2))
```



## 7.5 Typesetting Racket Code

```
(require pict/code)
```
package: `pict-lib`

This library is re-provided by `slideshow/code`, but initializes `get-current-code-font-size`.

```
(typeset-code stx) → pict?
  stx : syntax?
```

Produces a pict for code in the given syntax object. The source-location information of the syntax object determines the line breaks, line indenting, and space within a row. Empty rows are ignored.

Beware that if you use `read-syntax` on a file port, you may have to turn on line counting via `port-count-lines!` for the code to typeset properly.

Normally, `typeset-code` is used through the `code` syntactic form, which works properly with compilation, and that escapes to pict-producing code via `unsyntax`. See also `define-code`.

Embedded picts within *stx* are used directly. Row elements are combined using and operator like `htl-append`, so use `code-align` (see below) as necessary to add an ascent to ascentless picts. More precisely, creation of a line of code uses `pict-last` to determine the end point of the element most recently added to a line; the main effect is that closing parentheses are attached in the right place when a multi-line pict is embedded in *stx*.

An identifier that starts with `_` is italicized in the pict, and the `_` is dropped, unless the `code-italic-underscore-enabled` parameter is set to `#f`. Also, unless `code-scripts-enabled` is set to `#f`, `_` and `^` in the middle of a word create superscripts and subscripts, respectively (like TeX); for example `foo^4_ok` is displayed as the identifier `foo` with a `4` superscript and an `ok` subscript.

Further, uses of certain identifiers in *stx* typeset specially:

- `code:blank` — produces a space.
- `(code:comment s ...)` — produces a comment (prefixed with a single semicolon), with each *s* on the same line, where each *s* must be a string or a value that is pict convertible.
- `(code:comment2 s ...)` — produces a comment (prefixed with two semicolons), with each *s* on the same line, where each *s* must be a string or a value that is pict convertible.
- `(code:line datum ...)` — typesets the *datum* sequence, which is mostly useful for the top-level sequence, since `typeset-code` accepts only one argument.
- `(code:contract datum ...)` — like `code:line`, but every *datum* is colored as a comment, and a `;` is prefixed to every line.
- `(code:template datum ...)` — like `code:line`, but a `;` is prefixed to every line.
- `$` — typesets as a vertical bar (for no particularly good reason).

```
(code datum ...)
```

The macro form of typeset-code. Within a *datum*, unsyntax can be used to escape to an expression, and identifiers bound as syntax to code transformers trigger transformations.

For more information, see typeset-code and define-code, since code is defined as

```
(define-code code typeset-code)
```

Examples:

```
> (code (+ 1 2))
(+ 1 2)
> (code (+ 1 #,(+ 1 1)))
(+ 1 2)
> (code (+ 1 #,(frame (code 2))))
(+ 1 2)
> (define-syntax two (make-code-transformer #'(code 2)))
> (code (+ 1 two))
(+ 1 2)
```

```
(current-code-font) → text-style/c
(current-code-font style) → void?
  style : text-style/c
```

Parameter for a base font used to typeset text. The default is `(bold . modern). For even more control, see current-code-tt.

```
(current-code-tt) → (string? . -> . pict?)
(current-code-tt proc) → void?
  proc : (string? . -> . pict?)
```

Parameter for a one-argument procedure to turn a string into a pict, used to typeset text. The default is

```
(lambda (s) (text s (current-code-font) ((get-current-code-font-
size))))
```

This procedure is not used to typeset subscripts or other items that require font changes, where current-code-font is used directly.

```
(get-current-code-font-size) → (-> exact-nonnegative-integer?)
(get-current-code-font-size proc) → void?
  proc : (-> exact-nonnegative-integer?)
```

A parameter used to access the default font size. The slideshow/code library initializes this parameter to current-font-size.

```
(current-code-line-sep) → real?
(current-code-line-sep amt) → void?
  amt : real?
```

A parameter that determines the spacing between lines of typeset code. The default is 2.

```
(current-comment-color) → (or/c string? (is-a?/c color%))
(current-comment-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

A parameter for the color of comments.

```
(current-keyword-color) → (or/c string? (is-a?/c color%))
(current-keyword-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

A parameter for the color of syntactic-form names. See current-keyword-list.

```
(current-id-color) → (or/c string? (is-a?/c color%))
(current-id-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

A parameter for the color of identifiers that are not syntactic form names or constants.

```
(current-literal-color) → (or/c string? (is-a?/c color%))
(current-literal-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

A parameter for the color of literal values, such as strings and numbers. See also current-literal-list

```
(current-const-color) → (or/c string? (is-a?/c color%))
(current-const-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

A parameter for the color of constant names. See current-const-list.

```
(current-base-color) → (or/c string? (is-a?/c color%))
(current-base-color color) → void?
  color : (or/c string? (is-a?/c color%))
```

A parameter for the color of everything else.

```
(current-reader-forms) → (listof symbol?)
(current-reader-forms syms) → void?
  syms : (listof symbol?)
```

Parameter for a list of symbols indicating which built-in reader forms should be used. The default is `''quasiquote`. Remove a symbol to suppress the corresponding reader output.

```
(code-align pict) → pict?
  pict : pict?
```

Adjusts the ascent of *pict* so that its bottom aligns with the baseline for text; use this function when *pict* has no ascent.

```
(current-keyword-list) → (listof string?)
(current-keyword-list names) → void?
  names : (listof string?)
```

A list of strings to color as syntactic-form names. The default includes all of the forms provided by `racket/base` and all of the forms provided by `mzscheme`.

```
(current-const-list) → (listof string?)
(current-const-list names) → void?
  names : (listof string?)
```

A list of strings to color as constant names. The default is `null`.

```
(current-literal-list) → (listof string?)
(current-literal-list names) → void?
  names : (listof string?)
```

A list of strings to color as literals, in addition to literals such as strings. The default is `null`.

```
racket/base-const-list : (listof string?)
```

A list of strings that could be used to initialize the `current-const-list` parameter.

```
mzscheme-const-list : (listof string?)
```

A list of strings that could be used to initialize the `current-const-list` parameter.

```
(code-colorize-enabled) → boolean?
(code-colorize-enabled on?) → void?
  on? : any/c
```

A parameter to enable or disable all code coloring. The default is `#t`.

```
(code-colorize-quote-enabled) → boolean?
(code-colorize-quote-enabled on?) → void?
  on? : any/c
```

A parameter to control whether the datum under a `quote` is colorized as a literal (as in this documentation). The default is `#t`.

```
(code-italic-underscore-enabled) → boolean?
(code-italic-underscore-enabled on?) → void?
  on? : any/c
```

A parameter to control whether `_`-prefixed identifiers are italicized (dropping the `_`). The default is `#t`.

```
(code-scripts-enabled) → boolean?
(code-scripts-enabled on?) → void?
  on? : any/c
```

A parameter to control whether TeX-style subscripts and subscripts are recognized in an identifier.

```
(define-code code-id typeset-code-id)
(define-code code-id typeset-code-id escape-id)
```

Defines *code-id* as a macro that uses *typeset-code-id*, which is a function with the same input as `typeset-code`. The *escape-id* form defaults to `unsyntax`.

The resulting *code-id* syntactic form takes a sequence of *datum*s:

```
(code-id datum ...)
```

It produces a pict that typesets the sequence. Source-location information for the *datum* determines the layout of code in the resulting pict. The *code-id* is expanded in such a way that source location is preserved during compilation (so *typeset-code-id* receives a syntax object with source locations intact).

If a *datum* contains `(`*escape-id expr*`)`—perhaps as `#,`*expr* when *escape-id* is `unsyntax`—then the *expr* is evaluated and the result datum is spliced in place of the *escape-id* form in *datum*. If the result is not a syntax object, it is given the source location of the *escape-id* form. A pict value injected this way as a *datum* is rendered as itself.

If a *datum* contains `(transform-id` *datum* `...)` or `transform-id` for a `transform-id` that is bound as syntax to a code transformer, then the `(transform-id` *datum* `...)` or `transform-id` may be replaced with an escaped expression, depending on the code transformer's result.

```
(make-code-transformer proc-or-stx) → code-transformer?
  proc-or-stx : (or/c (syntax? . -> . (or/c syntax? #f))
                      syntax?)
prop:code-transformer : struct-type-property?
(code-transformer? v) → boolean?
  v : any/c
```

Exported `for-syntax` for creating *code transformers*.

For code transformer created with `(make-code-transformer proc)`, `proc` takes a syntax object representing the use of an identifier bound to the transformer, and it produces an expression whose value replaces the identifier use within a `code` form or a form defined via `define-code`. Like a macro transformer, a code transformer is triggered either by a use of the bound identifier in an "application" position, in which case the transformer receives the entire "application" form, or the identifier by itself can also trigger the transformer. The code transformer's *proc* can return `#f`, in which case the use of the identifier is left untransformed; if the identifier was used in an "application" position, the transformer *proc* will be called again for the identifier use by itself.

A code transformer produced by `(make-code-transformer stx)` is equivalent to

```
(make-code-transformer (lambda (use-stx)
                         (if (identifier? use-stx)
                             stx
                             #f)))
```

A structure type with the `prop:code-transformer` property implements a code transformer. The property value must be a procedure of one argument, which receives the structure and returns a procedure that is like a *proc* passed to `make-code-transformer`, except that the property value takes the structure instance as an argument before the syntax object to transform.

The `code-transformer?` predicate returns `#t` for a value produced by `make-code-transformer` or for an instance of a structure type with the `prop:code-transformer` property, `#f` otherwise.

Example:

```
> (let-syntax ([bag (make-code-transformer #'(code hat))]
               [copy (make-code-transformer (syntax-rules ()
                                              [(_ c) (code (* 2 c))]))])
    (inset (frame (code ((copy cat) in the bag))) 2))
((* 2 cat) in the hat)
```

```
(define-exec-code (pict-id runnable-id string-id)
  datum ...)
```

Binds *pict-id* to the result of `(code` *datum* `...)`, except that if an identifier `_` appears anywhere in a *datum*, then the identifier and the following expression are not included for `code`.

Meanwhile, *runnable-id* is bound to a syntax object that wraps the *datum*s in a `begin`. In this case, `_`s are removed from the *datum*s, but not the following expression. Thus, an `_` identifier is used to comment out an expression from the pict, but have it present in the syntax object for evaluation.

The *string-id* is bound to a string representation of the code that is in the pict. This string is useful for copying to the clipboard with `(send the-clipboard set-clipboard-string string-id 0)`.

```
(define-exec-code/scale scale-expr (pict-id runnable-id string-id)
  datum ...)
```

Like `define-exec-code`, but with a scale to use via `scale/improve-new-text` when generating the pict.

```
comment-color : (or/c string? (is-a?/c color%))
keyword-color : (or/c string? (is-a?/c color%))
id-color : (or/c string? (is-a?/c color%))
literal-color : (or/c string? (is-a?/c color%))
```

For backward compatibility, the default values for `current-comment-color`, etc.

```
(code-pict-bottom-line-pict pict) → (or/c pict? #f)
  pict : pict?
```

The same as `pict-last`, provided for backward compatibility.

```
(pict->code-pict pict bl-pict) → pict?
  pict : pict?
  bl-pict : (or/c pict? #f)
```

Mainly for backward compatibility: returns `(if bl-pict (use-last pict (or (pict-last bl-pict) bl-pict)))`.

### 7.5.1 Codeblocks

```
(codeblock-pict  code-string
                 [#:keep-lang-line? keep?]) → pict?
  code-string : string?
  keep? : any/c = #t
```

Like Scribble's `codeblock` but generates picts.

Unlike `code`, `codeblock-pict` formats and colors based on the code's language's lexer, which makes it suitable for use with code that doesn't use Racket syntax.

When *keep?* is `#f`, the first `#lang` line is removed from the pict.

Example:

```
> (codeblock-pict
    #:keep-lang-line? #f
    (string-join
      '("#lang 2d racket"
        "(require 2d/cond)"
        "(define (same? a b)"
        "  #2dcond"
        "  ╔═══════════╦═══════════════════════════╦═══════════╗"
        "  ║           ║         (pair? a)         ║ (number? a) ║"
        "  ╠═══════════╬═══════════════════════════╬═══════════╣"
        "  ║ (pair? b) ║ (and (same? (car a)       ║     #f    ║"
        "  ║           ║              (car b))     ║           ║"
        "  ║           ║      (same? (cdr a)       ║           ║"
        "  ║           ║             (cdr b)))     ║           ║"
        "  ╠═══════════╬═══════════════════════════╬═══════════╣"
        "  ║ (number? b) ║          #f             ║   (= a b)   ║"
        "  ╚═══════════╩═══════════════════════════╩═══════════╝)")
      "\n"))
```

```
(require 2d/cond)
(define (same? a b)
  #2dcond
```

| | (pair? a) | (number? a) |
|---|---|---|
| (pair? b) | (and (same? (car a) (car b)) (same? (cdr a) (cdr b))) | #f |
| (number? b) | #f | (= a b) |

```
)
```

```
(current-token-class->color)
 → (-> symbol? (or/c string? (is-a?/c color%)))
(current-token-class->color to-color) → void?
  to-color : (-> symbol? (or/c string? (is-a?/c color%)))
```

A parameter that provides additional control over the colors used by `codeblock-pict`. The *to-color* procedure is applied to each token class produced by a language's lexer to map the class name to a particular color.

The default value maps the standard set of token classes (as recognized by DrRacket) to the corresponding colors used by `code`. Specifically, it maps

- `'no-color`, `'parenthesis`, and `'hash-colon-keyword` to `(current-base-color)`,
- `'symbol` to `(current-id-color)`,
- `'keyword` to `(current-keyword-color)`,
- `'string` and `'constant` to `(current-literal-color)`,
- `'comment` to `(current-comment-color)`,
- `'white-space` to `"white"`,
- and everything else to `"black"`.

It is usually sufficient to adjust the individual color parameters also used by `code`, but `current-token-class->color` allows for more fine-grained control over how `codeblock-pict` interprets token classes.

Added in version 1.8 of package `pict-lib`.

# 8 Animation Helpers

These functions are designed to work with the slide constructors in `slideshow/play`.

## 8.1 Pict Interpolations

```
(fade-pict  n
            p1
            p2
           [#:combine combine
            #:composite? composite?]) → pict?
  n : (real-in 0.0 1.0)
  p1 : pict?
  p2 : pict?
  combine : (pict? pict? . -> . pict?) = cc-superimpose
  composite? : any/c = #t
```

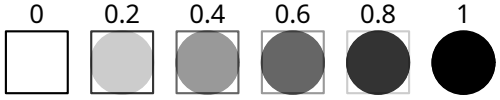
Interpolates *p1* and *p2*, where the result with *n* as `0.0` is *p1*, and the result with *n* as `1.0` is *p2*. For intermediate points, *p1* fades out while *p2* fades in as *n* changes from `0.0` to `1.0`. At the same time, the width and height of the generated pict are intermediate between *p1* and *p2*, and the relative baselines and last pict correspondingly morph within the bounding box.

The *combine* argument determines how *p1* and *p2* are aligned for morphing. For example, if *p1* and *p2* both contain multiple lines of text with the same line height but different number of lines, then using `ctl-superimpose` would keep the ascent line in a fixed location relative to the top of the resulting pict as the rest of the shape morphs around it.

The *composite?* argument is passed to the `cellophane` as used to implement fading.

Examples:

```
> (define (do-fade n)
    (fade-pict n (rectangle 30 30) (disk 30)))
> (apply ht-append 10
         (for/list ([n (in-range 0 1.2 0.2)])
           (vc-append (text (~r n #:precision 2))
                      (do-fade n))))
```




Changed in version 1.15 of package `pict-lib`: Added `#:composite?` and enabled compositing by default.

```
(fade-around-pict  n
                   p1
                   make-p2
                  [#:composite? composite?]) → pict?
  n : (real-in 0.0 1.0)
  p1 : pict?
  make-p2 : (pict? . -> . pict?)
  composite? : any/c = #t
```

Similar to `fade-pict`, but the target is not a fixed *p2*, but instead a function *make-p2* that takes a `laundered` `ghost` of *p1* and places it into a larger scene. Also, *p1* does not fade out as *n* increases; instead, *p1* is placed wherever its ghost appears in the result of *make-p2*.

For example,

```
> (get-current-code-font-size (λ () 20))
> (define do-fade
    (lambda (n)
      (fade-around-pict n
                        (code x)
                        (lambda (g) (code (+ #,g 1))))))
> (apply ht-append 10
         (for/list ([n (in-range 0 1.2 0.2)])
           (vc-append (text (~r n #:precision 2))
                      (do-fade n))))
```

animates the wrapping of `x` with a `(+ .... 1)` form.

Changed in version 1.15 of package `pict-lib`: Added `#:composite?` and enabled compositing by default.

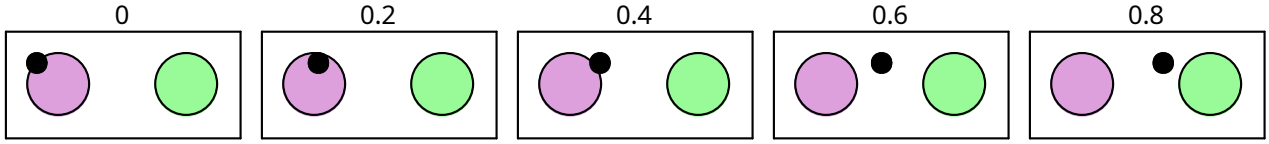
```
(slide-pict base p p-from p-to n) → pict?
  base : pict?
  p : pict?
  p-from : pict?
  p-to : pict?
  n : (real-in 0.0 1.0)
```

Pins *p* onto *base*, sliding from *p-from* to *p-to* (which are picts within *base*) as *n* goes from `0.0` to `1.0`. The top-left locations of *p-from* and *p-to* determine the placement of the top-left of *p*.

The *p-from* and *p-to* picts are typically `laundered` `ghosts` of *p* within *base*, but they can be any picts within *base*.

Examples:

```
> (define (do-slide n)
    (define p1 (disk 30 #:color "plum"))
    (define p2 (disk 30 #:color "palegreen"))
    (define p3 (frame (inset (hc-append 30 p1 p2) 10)))
    (slide-pict p3
                (disk 10)
                p1 p2 n))
> (apply ht-append 10
         (for/list ([n (in-range 0 1.2 0.2)])
           (vc-append (text (~r n #:precision 2))
                      (do-slide n))))
```

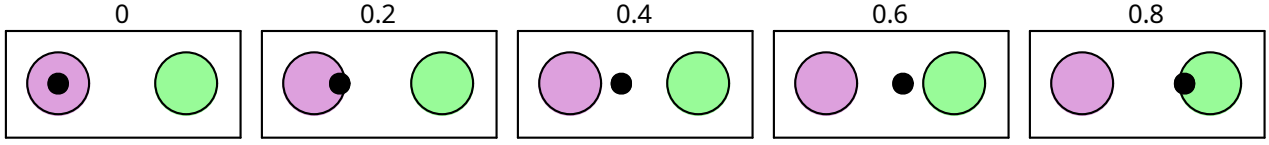



```
(slide-pict/center base p p-from p-to n) → pict?
  base : pict?
  p : pict?
  p-from : pict?
  p-to : pict?
  n : (real-in 0.0 1.0)
```

Like `slide-pict`, but aligns the center of *p* with *p-from* and *p-to*.

Examples:

```
> (define (do-slide n)
    (define p1 (disk 30 #:color "plum"))
    (define p2 (disk 30 #:color "palegreen"))
    (define p3 (frame (inset (hc-append 30 p1 p2) 10)))
    (slide-pict/center p3
                       (disk 10)
                       p1 p2 n))
> (apply ht-append 10
         (for/list ([n (in-range 0 1.2 0.2)])
           (vc-append (text (~r n #:precision 2))
                      (do-slide n))))
```



## 8.2 Merging Animations

```
(sequence-animations gen ...) → (-> (real-in 0.0 1.0) pict?)
  gen : (-> (real-in 0.0 1.0) pict?)
```

Converts a list of *gen* functions into a single function that uses each *gen* in sequence.

```
(reverse-animations gen ...) → (-> (real-in 0.0 1.0) pict?)
  gen : (-> (real-in 0.0 1.0) pict?)
```

Converts a list of *gen* functions into a single function that run `(sequence-animations gen ...)` in reverse.

## 8.3 Stretching and Squashing Time

```
(fast-start n) → (real-in 0.0 1.0)
  n : (real-in 0.0 1.0)
(fast-end n) → (real-in 0.0 1.0)
  n : (real-in 0.0 1.0)
(fast-edges n) → (real-in 0.0 1.0)
  n : (real-in 0.0 1.0)
(fast-middle n) → (real-in 0.0 1.0)
  n : (real-in 0.0 1.0)
```

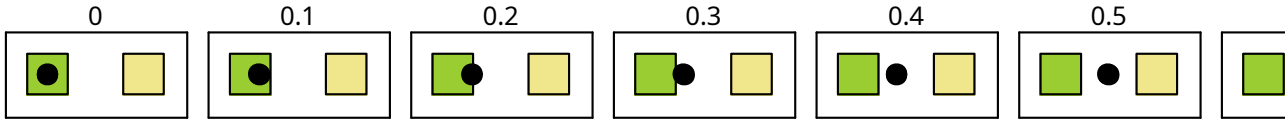
Monotonically but non-uniformly maps *n* with fixed points at `0.0` and `1.0`.

Suppose that we have the following definitions for our examples:

```
> (define (do-slide n fast-proc)
    (define p1 (filled-rectangle 20 20 #:color "yellowgreen"))
    (define p2 (filled-rectangle 20 20 #:color "khaki"))
    (define p3 (frame (inset (hc-append 25 p1 p2) 10)))
    (slide-pict/center
     p3
     (disk 10)
     p1 p2
     ; note use of fast-proc
     (fast-proc n)))
> (define (run-animation fast-proc)
    (apply ht-append 10
           (for/list ([n (in-range 0 1.09 0.1)])
             (vc-append (text (~r n #:precision 2))
                        (do-slide n fast-proc)))))
```

A normal use of the animation looks like this:

```
> (run-animation (λ (n) n))
```



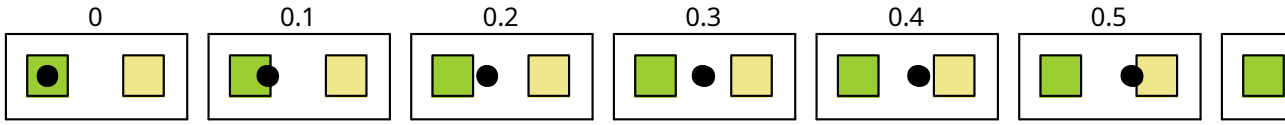

The `fast-start` mapping is convex, so that

```
(slide-pict base p p1 p2 (fast-start n))
```

appears to move quickly away from *p1* and then slowly as it approaches *p2*, assuming that *n* increases uniformly.

Applying it to the animation above produces this:

```
> (run-animation fast-start)
```

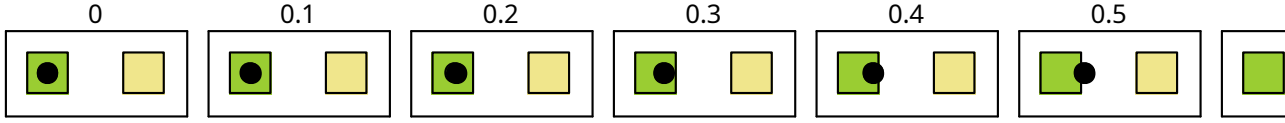



The `fast-end` mapping is concave, so that

```
(slide-pict base p p1 p2 (fast-end n))
```

appears to move slowly away from *p1* and then quickly as it approaches *p2*, assuming that *n* increases uniformly.

```
> (run-animation fast-end)
```

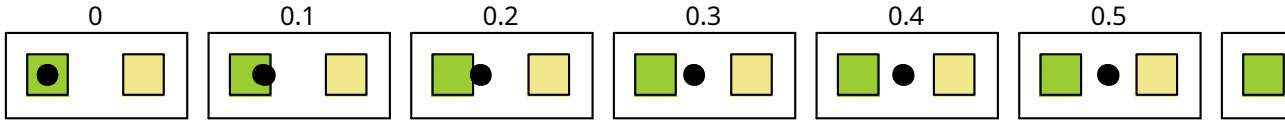



The `fast-edges` mapping is convex at first and concave at the end, so that

```
(slide-pict base p p1 p2 (fast-edges n))
```

appears to move quickly away from *p1*, then more slowly, and then quickly again near *p2*, assuming that *n* increases uniformly.

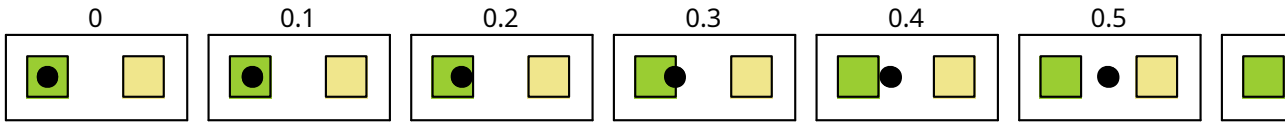
```
> (run-animation fast-edges)
```



The `fast-middle` mapping is concave at first and convex at the end, so that

```
(slide-pict base p p1 p2 (fast-middle n))
```

```
> (run-animation fast-middle)
```



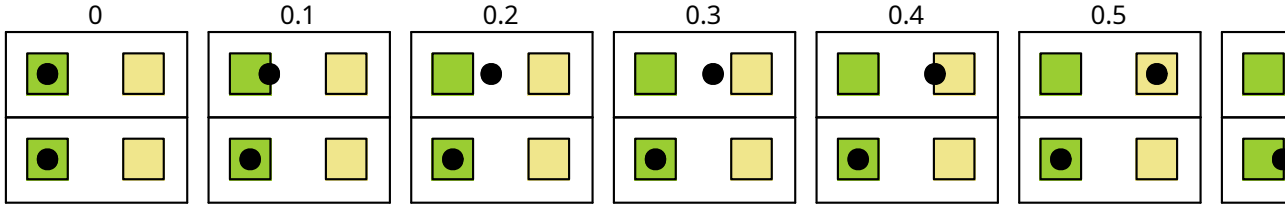

appears to move slowly away from *p1*, then more quickly, and then slowly again near *p2*, assuming that *n* increases uniformly.

```
(split-phase n) → (real-in 0.0 1.0) (real-in 0.0 1.0)
  n : (real-in 0.0 1.0)
```

Splits the progression of *n* from `0.0` to `1.0` into a progression from `(values 0.0 0.0)` to `(values 1.0 0.0)` and then `(values 1.0 0.0)` to `(values 1.0 1.0)`.

Here is an example that shows how to apply `split-phase` to the animation from the examples for `fast-start`:

```
> (apply ht-append 10
         (for/list ([n (in-range 0 1.09 0.1)])
           (define-values (n1 n2) (split-phase n))
           (vc-append (text (~r n #:precision 2))
                      (do-slide n1 (λ (n) n))
                      (do-slide n2 (λ (n) n)))))
```



# 9 Color Helpers

```
(require pict/color)
```
package: pict-lib

```
(red pict) → pict?
  pict : pict?
(orange pict) → pict?
  pict : pict?
(yellow pict) → pict?
  pict : pict?
(green pict) → pict?
  pict : pict?
(blue pict) → pict?
  pict : pict?
(purple pict) → pict?
  pict : pict?
(black pict) → pict?
  pict : pict?
(brown pict) → pict?
  pict : pict?
(gray pict) → pict?
  pict : pict?
(white pict) → pict?
  pict : pict?
(cyan pict) → pict?
  pict : pict?
(magenta pict) → pict?
  pict : pict?
```

These functions apply appropriate colors to picture p.

Example:

```
> (red (disk 20))
```

Added in version 1.4 of package pict-lib.

```
(light color) → color/c
  color : color/c
(dark color) → color/c
  color : color/c
```

These functions produce lighter or darker versions of a color.

Example:

```
> (hc-append (colorize (disk 20) "red")
             (colorize (disk 20) (dark "red"))
             (colorize (disk 20) (light "red")))
```

Added in version 1.4 of package `pict-lib`.

### color/c : flat-contract?

This contract recognizes color strings, `color%` instances, and RGB color lists.

Added in version 1.4 of package `pict-lib`.

# 10 Shadows

```
(require pict/shadow)
```
package: pict-lib

These pict transformations add shadows or blurring in various shapes and forms.

```
(blur p h-radius [v-radius]) → pict?
  p : pict?
  h-radius : (and/c real? (not/c negative?))
  v-radius : (and/c real? (not/c negative?)) = h-radius
```

Blurs *p* using an iterated box blur that approximates a gaussian blur. The *h-radius* and *v-radius* arguments control the strength of the horizontal and vertical components of the blur, respectively. They are given in terms of pict units, which may not directly correspond to screen pixels.

The `blur` function takes work proportional to

```
(* (pict-width p) (pict-height p))
```

but it may be sped up by a factor of up to `(processor-count)` due to the use of `future`s.

Examples:

```
> (blur (text "blur" null 40) 5)
```

```
> (blur (text "more blur" null 40) 10)
```

```
> (blur (text "much blur" null 40) 20)
```

```
> (blur (text "horiz. blur" null 40) 10 0)
```

The resulting pict has the same bounding box as *p*, so when picts are automatically `clip`ped (as in Scribble documents), the pict should be `inset` by the blur radius.

Example:

```
> (inset (blur (text "more blur" null 40) 10) 10)
```

Added in version 1.4 of package `pict-lib`.

```
(shadow  p
         radius
        [dx
         dy
         #:color color
         #:shadow-color shadow-color]) → pict?
  p : pict?
  radius : (and/c real? (not/c negative?))
  dx : real? = 0
  dy : real? = dx
  color : (or/c #f string? (is-a?/c color%)) = #f
  shadow-color : (or/c #f string? (is-a?/c color%)) = #f
```

Creates a shadow effect by superimposing *p* over a blurred version of *p*. The shadow is offset from *p* by (*dx*, *dy*) units.

If *color* is not `#f`, the foreground part is `(colorize p color)`; otherwise it is just *p*. If *shadow-color* is not `#f`, the shadow part is produced by blurring `(colorize p shadow-color)`; otherwise it is produced by blurring *p*.

The resulting pict has the same bounding box as *p*.

Examples:

```
> (inset (shadow (text "shadow" null 50) 10) 10)
```

```
> (inset (shadow (text "shadow" null 50) 10 5) 10)
```

```
> (inset (shadow (text "shadow" null 50)
                 5 0 2 #:color "white" #:shadow-color "red")
         10)
```

Added in version 1.4 of package `pict-lib`.

```
(shadow-frame  pict
               ...
              [#:sep separation
               #:margin margin
               #:background-color bg-color
               #:frame-color frame-color
               #:frame-line-width frame-line-width
               #:shadow-side-length shadow-side-length
               #:shadow-top-y-offset shadow-top-y-offset
               #:shadow-bottom-y-offset shadow-bottom-y-offset
               #:shadow-descent shadow-descent
               #:shadow-alpha-factor shadow-alpha-factor
               #:blur blur-radius])
 → pict?
  pict : pict?
  separation : real? = 5
  margin : real? = 20
  bg-color : (or/c string? (is-a?/c color%)) = "white"
  frame-color : (or/c string? (is-a?/c color%)) = "gray"
  frame-line-width : (or/c real? #f 'no-frame) = 0
  shadow-side-length : real? = 4
  shadow-top-y-offset : real? = 10
  shadow-bottom-y-offset : real? = 4
  shadow-descent : (and/c real? (not/c negative?)) = 40
  shadow-alpha-factor : real? = 3/4
  blur-radius : (and/c real? (not/c negative?)) = 20
```

Surrounds the *pict*s with a rectangular frame that casts a symmetric “curled paper” shadow.

The *picts* are vertically appended with *separation* space between them. They are placed on a rectangular background of solid *bg-color* with *margin* space on all sides. A frame of *frame-color* and *frame-line-width* is added around the rectangle, unless *frame-line-width* is `'no-frame`. The rectangle casts a shadow that extends *shadow-side-length* to the left and right, starts *shadow-top-y-offset* below the top of the rectangle and extends to *shadow-bottom-y-offset* below the bottom of the rectangle in the center and an additional *shadow-descent* below that on the sides. The shadow is painted using a linear gradient; *shadow-alpha-factor* determines its density at the center. Finally, the shadow is blurred by *blur-radius*; all previous measurements are pre-blur measurements.

Example:

```
> (shadow-frame (text "text in a nifty frame" null 60))
```

Added in version 1.4 of package `pict-lib`.

# 11 Conditional Combinations

```
(require pict/conditional)
```
package: pict-lib

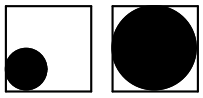
These pict control flow operators decide which pict of several to use. All branches are evaluated; the resulting pict is a combination of the pict chosen by normal conditional flow with ghost applied to all the other picts. The result is a picture large enough to accommodate each alternative, but showing only the chosen one. This is useful for staged slides, as the pict chosen may change with each slide but its size and position will not.

```
(pict-if maybe-combine test-expr then-expr else-expr)

maybe-combine =
              | #:combine combine-expr
```

Chooses either *then-expr* or *else-expr* based on *test-expr*, similarly to if. Combines the chosen, visible image with the other, invisible image using *combine-expr*, defaulting to lbl-superimpose.

Example:

```
> (let ([f (lambda (x)
             (pict-if x
                      (disk 20)
                      (disk 40)))])
    (hc-append 10
               (frame (f #t))
               (frame (f #f))))
```



Added in version 1.4 of package pict-lib.

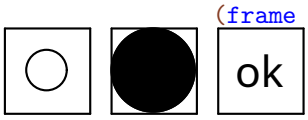
```
(pict-cond maybe-combine [test-expr pict-expr] ...)

maybe-combine =
              | #:combine combine-expr
```

Chooses a *pict-expr* based on the first successful *test-expr*, similarly to cond. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to lbl-superimpose.

Example:

```
> (let ([f (lambda (x)
```

```
            (pict-cond #:combine cc-superimpose
              [(eq? x 'circle) (circle 20)]
              [(eq? x 'disk) (disk 40)]
              [(eq? x 'text) (text "ok" null 20)])))])
    (hc-append 10
               (frame (f 'circle))
               (frame (f 'disk))
               (frame (f 'text))))
```



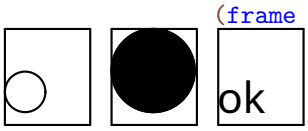

Added in version 1.4 of package `pict-lib`.

```
(pict-case test-expr maybe-combine [literals pict-expr] ...)

maybe-combine =
              | #:combine combine-expr
```

Chooses a *pict-expr* based on *test-expr* and each list of *literals*, similarly to `case`. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to `lbl-superimpose`.

Example:

```
> (let ([f (lambda (x)
             (pict-case x
               [(circle) (circle 20)]
               [(disk) (disk 40)]
               [(text) (text "ok" null 20)]))])
    (hc-append 10
               (frame (f 'circle))
               (frame (f 'disk))
               (frame (f 'text))))
```




Added in version 1.4 of package `pict-lib`.

```
(show pict [show?]) → pict?
  pict : pict?
  show? : any/c = #t
(hide pict [hide?]) → pict?
  pict : pict?
  hide? : any/c = #t
```

These functions conditionally show or hide an image, essentially choosing between *pict* and (ghost *pict*). The only difference between the two is the default behavior and the opposite meaning of the *show?* and *hide?* booleans. Both functions are provided for mnemonic purposes.

Added in version 1.4 of package `pict-lib`.

# 12 Tree Layout

These functions specify tree layouts and functions that render them as picts.

```
(require pict/tree-layout)      package: pict-lib
```

```
(tree-layout [#:pict node-pict] child ...) → tree-layout?
  node-pict : (or/c #f pict?) = #f
  child : (or/c tree-layout? tree-edge? #f)
```

Specifies an interior node of a tree for use with one of the renderers below.

If the children are tree-layout?s, then they have edges created by passing the corresponding tree-layout?s directly to tree-edge. Children that are #f correspond to leaf nodes that are not drawn.

The default *node-pict* (used when *node-pict* is #f) is .

```
(tree-edge  node
            [#:edge-color edge-color
             #:edge-width edge-width
             #:edge-style edge-style]) → tree-edge?
  node : (and/c tree-layout? (not/c #f))
  edge-color : (or/c string?                        = "gray"
                     (is-a?/c color%)
                     (list/c byte? byte? byte?))
  edge-width : (or/c 'unspecified real? #f) = 'unspecified
  edge-style : (or/c 'unspecified 'transparent 'solid 'xor 'hilite
                     'dot 'long-dash 'short-dash 'dot-dash
                     'xor-dot 'xor-long-dash 'xor-short-dash
                     'xor-dot-dash)
               = 'unspecified
```

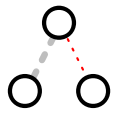
This function specifies an edge from some parent to the given *node*. It it intended to be used with tree-layout, on a non-#f node.

When *edge-width* is 'unspecified, the line width will not be set. This is intended to allow the line width to be set for the whole pict via linewidth. Otherwise, *edge-width* is interpreted the same way as the width argument for the linewidth function. *edge-style* behaves similarly, its argument interpreted as the style argument for the linestyle function.

Example:

```
> (naive-layered (tree-layout
                  (tree-edge #:edge-width 3 (tree-layout))
```

```
(tree-edge #:edge-color "red"
           #:edge-style 'dot
           (tree-layout)))
```



Changed in version 1.3 of package `pict-lib`: Added the `#:edge-width` option.
Changed in version 1.9: Added the `#:edge-style` option.

```
(tree-layout? v) → boolean?
  v : any/c
```

Recognizes a tree layout. It returns `#t` when given `#f` or the result of `tree-layout`.

```
(binary-tree-layout? v) → boolean?
  v : any/c
```

Recognizes a `tree-layout?` that represents a binary tree. That is, each interior node has either two children or is `#f`. Note that a node with zero children does not count as a leaf for the purposes of `binary-tree-layout?`.

Examples:

```
> (binary-tree-layout? (tree-layout #f #f))
#t
> (binary-tree-layout? #f)
#t
> (binary-tree-layout? (tree-layout (tree-layout) (tree-layout)))
#f
```

```
(tree-edge? v) → boolean?
  v : any/c
```

Recognizes an `tree-edge`.

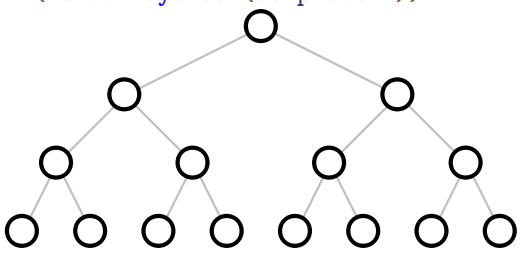
```
(naive-layered  tree-layout
               [#:x-spacing x-spacing
                #:y-spacing y-spacing
                #:transform transform]) → pict?
  tree-layout : tree-layout?
  x-spacing : (or/c (and/c real? positive?) #f) = #f
  y-spacing : (or/c (and/c real? positive?) #f) = #f
  transform : (-> real? real? (values real? real?)) = values
```

Uses a naive algorithm that ensures that all nodes at a fixed depth are the same vertical distance from the root (dubbed "layered"). It recursively lays out subtrees and then horizontally combines them, aligning them at their tops. Then it places the root node centered over the children nodes.

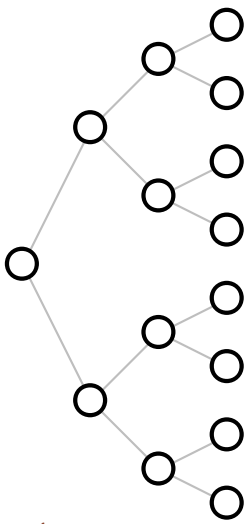
The *transform* argument applies a coordinate transformation to each of the nodes after it has been layed out. The bounding box of the resulting pict encompasses the corners of the original bounding box after the transformation has been applied to them.

Examples:

```
> (define (complete d)
    (cond
      [(zero? d) #f]
      [else (define s (complete (- d 1)))
            (tree-layout s s)]))
> (naive-layered (complete 4))
```

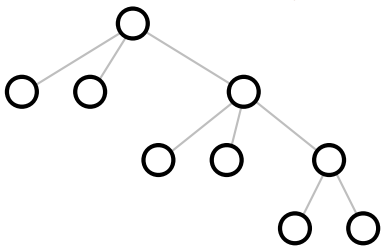


```
> (naive-layered (complete 4) #:transform (lambda (x y) (values y x)))
```

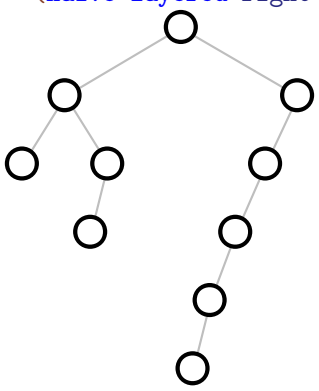


```
> (naive-layered (tree-layout
                  (tree-layout)
                  (tree-layout)
                  (tree-layout
```

```
                  (tree-layout)
                  (tree-layout)
                  (tree-layout
                   (tree-layout)
                   (tree-layout)))))
```



```
> (define right-subtree-with-left-chain
    (tree-layout
     (tree-layout
      (tree-layout #f #f)
      (tree-layout
       (tree-layout #f #f)
       #f))
     (tree-layout
      (tree-layout
       (tree-layout
        (tree-layout
         (tree-layout #f #f)
         #f)
        #f)
       #f)
      #f)))
> (naive-layered right-subtree-with-left-chain)
```



Changed in version 1.13 of package `pict-lib`: Added the `#:transform` option.

```
(binary-tidier  tree-layout
                [#:x-spacing x-spacing
                 #:y-spacing y-spacing
                 #:transform transform]) → pict?
  tree-layout : binary-tree-layout?
  x-spacing : (or/c (and/c real? positive?) #f) = #f
  y-spacing : (or/c (and/c real? positive?) #f) = #f
  transform : (-> real? real? (values real? real?)) = values
```

Uses the layout algorithm from *Tidier Drawing of Trees* by Edward M. Reingold and John S. Tilford (IEEE Transactions on Software Engineering, Volume 7, Number 2, March 1981) to lay out *tree-layout*.

The layout algorithm guarantees a number of properties, namely:

- nodes at the same level of tree appear at the same vertical distance from the top of the pict
- parents are centered over their children, which are placed from left to right,
- isomorphic subtrees are drawn the same way, no matter where they appear in the complete tree, and
- a tree and its mirror image produce picts that are mirror images of each other (which also holds for subtrees of the complete tree).

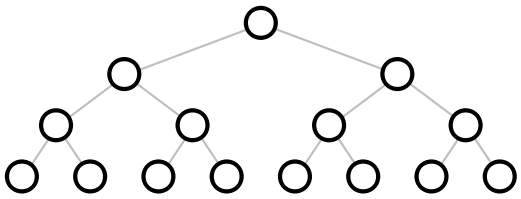
Within those constraints, the algorithm tries to make as narrow a drawing as it can, even to the point that one subtree of a given node might cross under the other one.

More precisely, it recursively lays out the two subtree and then, without adjusting the layout of the two subtrees, moves them as close together as it can, putting the root of the new tree centered on top of its children. (It does this in linear time, using clever techniques as discussed in the paper.)

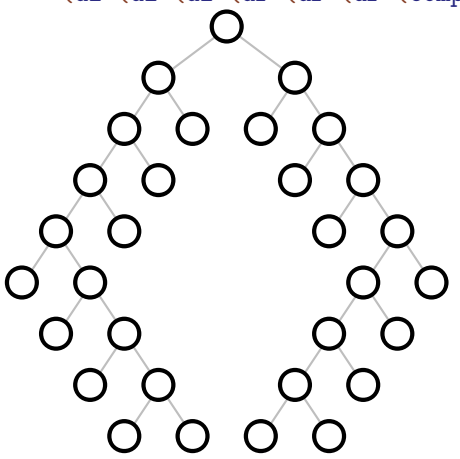
The *x-spacing* and *y-spacing* are the amount of space that each row and each column takes up, measured in pixels. If *x-spacing* is `#f`, it is the width of the widest node `pict?` in the tree. If *y-spacing* is `#f`, it is `1.5` times the width of the widest node `pict?` in the tree. The *transform* is the same as in `naive-layered`.

Examples:

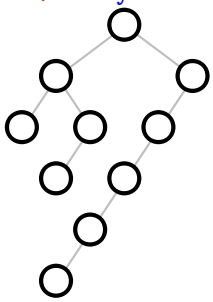
```
> (binary-tidier (complete 4))
```



```
> (define (dl t) (tree-layout (tree-layout #f #f) t))
> (define (dr t) (tree-layout t (tree-layout #f #f)))
> (binary-tidier
   (tree-layout
    (dr (dr (dr (dl (dl (dl (complete 2)))))))
    (dl (dl (dl (dr (dr (dr (complete 2)))))))))
```



```
> (binary-tidier right-subtree-with-left-chain)
```



Changed in version 1.13 of package `pict-lib`: Added the `#:transform` option.

```
(hv-alternating  tree-layout
                [#:x-spacing x-spacing
                 #:y-spacing y-spacing
                 #:transform transform]) → pict?
  tree-layout : binary-tree-layout?
```

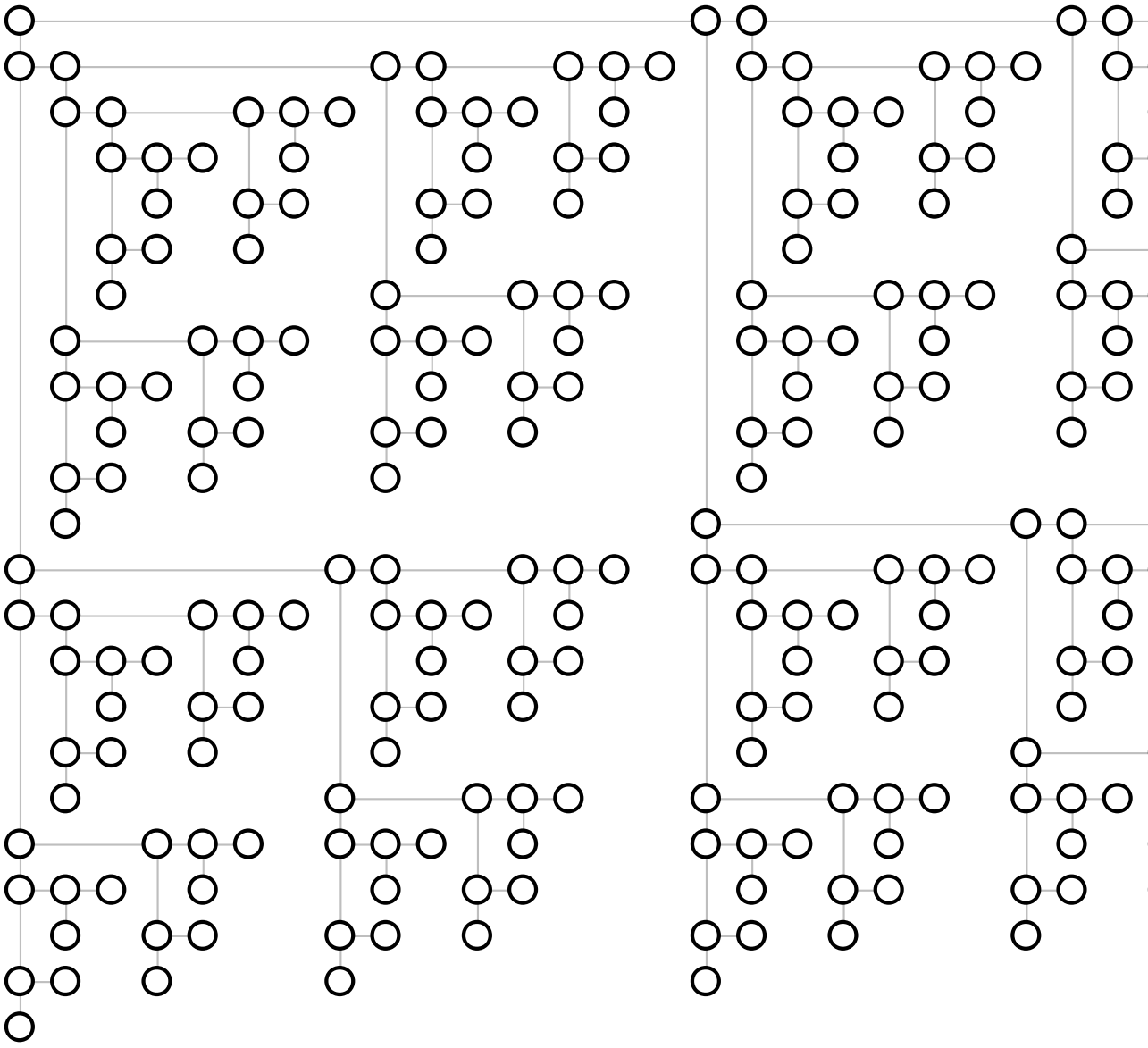
```
x-spacing : (or/c (and/c real? positive?) #f) = #f
y-spacing : (or/c (and/c real? positive?) #f) = #f
transform : (-> real? real? (values real? real?)) = values
```

Uses the “CT” binary tree layout algorithm from *A note on optimal area algorithms for upward drawing of binary trees* by P. Crescenzi, G. Di Battista, and A. Piperno (Computational Geometry, Theory and Applications, 1992) to lay out `tree-layout`.

It adds horizontal and vertical space between layers based on `x-spacing` and `y-spacing`. If either is `#f`, `1.5` times the size of the biggest node is used. The `transform` is the same as in `naive-layered`.

Example:

```
> (hv-alternating (complete 8))
```



Changed in version 1.13 of package `pict-lib`: Added the `#:transform` option.

Added in version 6.0.1.4 of package `pict-lib`.

# 13 Miscellaneous

```
(hyperlinkize pict) → pict?
  pict : pict-convertible?
```

Adds an underline and blue color. The *pict*'s height and descent are extended.

Example:

```
> (hyperlinkize (text "Help me I'm trapped in this documentation"
                      null 15))
```

Help me I'm trapped in this documentation

```
(scale-color factor color) → (is-a?/c color%)
  factor : real?
  color : (or/c string (is-a?/c color%))
```

Scales a color, making it brighter or darker. If the factor is less than 1, the color is darkened by multiplying the RGB components by the factor. If the factor is greater than 1, the color is lightened by dividing the gap between the RGB components and 255 by the factor.

Examples:

```
> (disk 30 #:color "royalblue")
```

```
> (disk 30 #:color (scale-color 0.5 "royalblue"))
```

```
> (disk 30 #:color (scale-color 1.5 "royalblue"))
```

```
(color-series dc
              max-step
              step-delta
              start
              end
              proc
              set-pen?
              set-brush?) → void?
  dc : (is-a?/c dc<%>)
  max-step : exact-nonnegative-integer?
```

```
step-delta : (and/c exact? positive?)
start : (or/c string? (is-a?/c color%))
end : (or/c string? (is-a?/c color%))
proc : (exact? . -> . any)
set-pen? : any/c
set-brush? : any/c
```

Calls a *proc* multiple times, gradually changing the pen and/or brush color for each call. For the first call, the current pen and/or brush color matches *start*; for the last call, it matches *end*; and for intermediate calls, the color is an intermediate color.

The *max-step* and *step-delta* arguments should be exact numbers; the procedure is called with each number from 0 to *max-step* inclusive using a *step-delta* increment.

# 14 Rendering

```
(dc-for-text-size) → (or/c #f (is-a?/c dc<%>))
(dc-for-text-size dc) → void?
  dc : (or/c #f (is-a?/c dc<%>))
```

A parameter that is used to determine the bounding box of picts created with `text`.

The drawing context installed in this parameter need not be the same as the ultimate drawing context, but it should measure text in the same way. Under normal circumstances, font metrics are the same for all drawing contexts, so the default value of `dc-for-text-size` is a `bitmap-dc%` that draws to a 1-by-1 bitmap.

```
(convert-bounds-padding)
 → (list/c (>=/c 0) (>=/c 0) (>=/c 0) (>=/c 0))
(convert-bounds-padding padding) → void?
  padding : (list/c (>=/c 0) (>=/c 0) (>=/c 0) (>=/c 0))
```

A parameter that determines an amount of padding added to each edge of a pict when converting to a format like `'png@2x-bytes+bounds8` (see `file/convertible`). The default value of the parameter is `'(3 3 3 3)`, which adds three pixels to each edge to accommodate a small amount of drawing outside the pict's bounding box.

Added in version 1.2 of package `pict-lib`.

```
(draw-pict pict dc x y) → void?
  pict : pict-convertible?
  dc : (is-a?/c dc<%>)
  x : real?
  y : real?
```

Draws *pict* to *dc*, with its top-left corner at offset (*x*, *y*).

```
(pict->bitmap  pict
               [smoothing
                #:make-bitmap make-a-bitmap]) → (is-a?/c bitmap%)
  pict : pict-convertible?
  smoothing : (or/c 'unsmoothed 'smoothed 'aligned) = 'aligned
  make-a-bitmap : (exact-positive-integer? exact-positive-integer?
                   . -> . (is-a?/c bitmap%))
                = make-bitmap
```

Returns a `bitmap%` with *pict* drawn on it in the top-left corner (`0`, `0`). Drawing the pict into the bitmap uses the smoothing mode *smoothing*; see `set-smoothing` for more information on smoothing modes.

The *make-a-bitmap* function is used to create the bitmap, giving it the size of *pict* rounded up to integer dimensions. The default function, make-bitmap, creates a bitmap with an alpha channel. Supply make-screen-bitmap, instead, to create a bitmap that is consistent with drawing directly to the screen.

Changed in version 1.7 of package pict-lib: Added the #:make-bitmap argument.

```
(pict->argb-pixels pict [smoothing]) → bytes?
  pict : pict-convertible?
  smoothing : (or/c 'unsmoothed 'smoothed 'aligned) = 'aligned
```

Returns the bytes? with the pixels corresponding the bitmap that pict->bitmap returns. Each pixel has four bytes in the result: the alpha, red, green, and blue components.

Examples:

```
> (pict->argb-pixels
   (filled-rectangle 1 1))
#"\377\0\0\0"
> (pict->argb-pixels
   (colorize (filled-rectangle 1 1) "red"))
#"\377\377\0\0"
```

Added in version 1.1 of package pict-lib.

```
(argb-pixels->pict bytes width) → pict?
  bytes : bytes?
  width : exact-nonnegative-integer?
```

Constructs a pict from *bytes* with the width *width*. Each pixel in the resulting pict corresponds to four entries in *bytes*: the alpha value, and the red, green, and blue values.

Example:

```
> (let ([b (make-bytes (* 40 40 4) 255)])
    (for ([x (in-range (bytes-length b))])
      ; when in one of two vertical bands (10-20 & 30-40)
      (when (or (<= 10 (modulo (quotient x 4) 40) 20)
                (<= 30 (modulo (quotient x 4) 40) 40))
        ; change the red and green fields of the pixel
        (when (= 1 (modulo x 4)) (bytes-set! b x 0))
        (when (= 2 (modulo x 4)) (bytes-set! b x 150))))
    (argb-pixels->pict b 40))
```

Added in version 1.1 of package pict-lib.

```
(make-pict-drawer pict)
 → ((is-a?/c dc<%>) real? real? . -> . void?)
  pict : pict-convertible?
```

Generates a pict-drawer procedure for multiple renderings of *pict*. Using the generated procedure can be faster than repeated calls to draw-pict.

```
(show-pict  pict
           [w
            h]
            #:frame-x frame-x
            #:frame-y frame-y
            #:frame-style frame-style) → void?
  pict : pict-convertible?
  w : (or/c #f exact-nonnegative-integer?) = #f
  h : (or/c #f exact-nonnegative-integer?) = #f
  frame-x : (or/c (integer-in -10000 10000) #f)
  frame-y : (or/c (integer-in -10000 10000) #f)
  frame-style : (listof (or/c 'no-resize-border 'no-caption
                              'no-system-menu 'hide-menu-bar
                              'toolbar-button 'float 'metal))
```

Opens a frame that displays *pict*. The frame adds one method, set-pict, which takes a pict to display. The optional *w* and *h* arguments specify a minimum size for the frame's drawing area, and the *frame-x*, *frame-y*, and *frame-style* keyword arguments behave in the same manner as x, y, and style arguments for the frame%.

```
(current-expected-text-scale) → (list/c real? real?)
(current-expected-text-scale scales) → void?
  scales : (list/c real? real?)
```

A parameter used to refine text measurements to better match an expected scaling of the image. The scale/improve-new-text form sets this parameter while also scaling the resulting pict.

# 15 Conversion to Picts

```
(require pict/convert)
```
package: pict-lib

The pict/convert library defines a protocol for values to be *pict convertible*, meaning they can convert themselves to picts. The protocol is used by DrRacket's interactions window, for example, to render values that it prints. Anything that is pict-convertible? can be used wherever a pict can be used. These values will be automatically converted to a pict when needed.

This set includes at least images from 2htdp/image, but is extensible so may include other values as well.

```
prop:pict-convertible : struct-type-property?
```

A property whose value should be a procedure matching the contract (-> any/c pict?). The procedure is called when a structure with the property is passed to pict-convert; the argument to the procedure is the structure, and the procedure's result should be a pict.

```
prop:pict-convertible? : struct-type-property?
```

A property whose value should be a predicate procedure (i.e., matching the contract predicate/c).

If this property is not set, then it is assumed to be the function (λ (x) #t).

If this property is set, then this procedure is called by pict-convertible? to determine if this particular value is convertible (thereby supporting situations where some instances of a given value are convertible to picts, but others are not).

```
(pict-convertible? v) → boolean?
  v : any/c
```

Returns #t if v are pict convertible (by having the prop:pict-convertible property) and #f otherwise. This function returns #t for pict?s.

```
(pict-convert v) → pict?
  v : pict-convertible?
```

Requests a data conversion from v to a pict.