

Slideshow: Figure and Presentation Tools

Version 5.3.6

Matthew Flatt
and Robert Bruce Findler

August 9, 2013

Slideshow is a library for creating presentation slides. Unlike Powerpoint, Slideshow provides no WYSIWYG interface for constructing slides. Instead, like SliTeX, a presentation is generated by a program.

To get started, run the `slideshow` executable, and click the Run Tutorial link.

To learn more about why Slideshow is cool, see also “Slideshow: Functional Presentations” [Findler06].

`#lang slideshow`

Most of the bindings defined in the manual are provided by the `slideshow` language, which also re-exports all of `racket` except for `printable<%>` (due to backward-compatibility issues).

Contents

1	Creating Slide Presentations	4
1.1	Slide Basics	4
1.2	Staging Slides	5
1.3	Display Size and Fonts	7
1.4	Command-line Options	7
1.5	Printing	8
2	Making Pictures	9
2.1	Pict Datatype	9
2.2	Basic Pict Constructors	11
2.3	Pict Combiners	22
2.4	Pict Drawing Adjusters	27
2.5	Bounding Box Adjusters	30
2.6	Pict Finders	31
2.7	More Pict Constructors	33
2.7.1	Dingbats	33
2.7.2	Balloon Annotations	38
2.7.3	Face	41
2.7.4	Flash	44
2.8	Miscellaneous	46
2.9	Rendering	47
2.10	Conversion to Picts	48
3	Making Slides	50
3.1	Primary Slide Functions	50
3.2	Slide Registration	54
3.3	Viewer Control	55
3.4	Constants and Layout Variables	56
3.5	Configuration	58
3.6	Pict-Staging Helper	60
3.7	Slides to Picts	61
4	Typesetting Racket Code	63
5	Animations	71
5.1	Generating Animated Slides	71
5.2	Animation Helpers	73
5.3	Merging Animations	74
5.4	Stretching and Squashing Time	75
	Bibliography	77
	Index	78

1 Creating Slide Presentations

The `slideshow` module acts as a language that includes:

- all of `racket`;
- pict-creating functions from `slideshow/pict`; and
- slide-composing functions from `slideshow/base`.

The `slideshow` and `slideshow/base` module initialization also check the `current-command-line-arguments` parameter to configure the slide mode (e.g., printing).

The rest of this section repeats information that is presented by the tutorial slideshow, which can be viewed by running the `slideshow` executable and clicking the Run Tutorial link.

1.1 Slide Basics

The main Slideshow function is `slide`, which adds a slide to the presentation with a given content. For example, the “Hello World” presentation can be defined by the following module:

```
#lang slideshow

(slide
 #:title "How to Say Hello"
 (t "Hello World!"))
```

The `t` function in this example creates a pict containing the given text using the default font and style.

Executing the above module pops up a slide-presentation window. Type Alt-q (or Meta-q) to end the slides. Here are more controls:

Alt-q, Meta-q, or Cmd-q	: end slide show
Esc	: if confirmed, end show
Right arrow, Space, f, n, or click	: next slide
Left arrow, Backspace, Delete, or b	: previous slide
g	: last slide
l	: first slide
s	: next slide with a different title/name
a	: previous slide starting different title/name
Alt-g, Cmd-g, or Meta-g	: select a slide
Alt-p, Cmd-p, or Meta-p	: show/hide slide number

Alt-c, Cmd-c, or Meta-c	: show/hide commentary
Alt-d, Cmd-d, or Meta-d	: show/hide preview
Alt-m, Cmd-m, or Meta-m	: show/hide mouse cursor
Shift with arrow	: move window 1 pixel
Alt, Meta, or Cmd with arrow	: move window 10 pixels

The `slide` function accepts any number of arguments. Each argument is a pict to be centered on the slide. The picts are stacked vertically with `(current-gap-size)` separation between each pict, and the total result is centered (as long as there's a gap of at least `(* 2 (current-gap-size))` between the title and content).

```
#lang slideshow

(slide
 #:title "How to Say Hello"
 (t "Hello World!")
 (t "Goodbye Drow!"))
```

Various functions format paragraphs and generate bulleted items for lists. For example, `item` creates a bulleted paragraph that spans (by default) the middle 2/3 of the slide:

```
#lang slideshow

(slide
 #:title "How to Say Hello"
 (item "If you want to create an example, you"
       "can always do something with" (bt "Hello World!"))
 (item "It's a bit silly, but a follow-up example"
       "could be" (bt "Goodbye Drow!")))
```

As the example illustrates, the `item` function accepts a mixture of strings and pict, and it formats them as a paragraph.

1.2 Staging Slides

The `slide` function creates a slide as a side effect. It can be put inside a function to abstract over a slide:

```
#lang slideshow

(define (slide-n n)
  (slide
   #:title "How to Generalize Slides"
   (item "This is slide number" (number->string n))))
```

```
(slide-n 1)
(slide-n 2)
(slide-n 3)
```

The `slide` function also has built-in support for some common multi-slide patterns. Each element argument to `slide` is usually a pict, but there are a few other possibilities:

- If an element is `'next`, then a slide is generated containing only the preceding elements, and then the elements are re-processed without the `'next`. Multiple `'next` elements generate multiple slides.
- If an element is `'alts`, then the next element must be a list of element lists. Each list up to the last one is appended to the elements before `'alts` and the resulting list of elements is processed. The last lists is appended to the preceding elements along with the remaining elements (after the list of lists) and the result is re-processed.
- A `'nothing` element is ignored (useful as a result of a branching expression).
- A `'next!` element is like `'next`, except that it is preserved when condensing (via the `--condense` flag).
- A `'alts~` element is like `'alts`, except that it is *not* preserved when condensing.
- A comment produced by `comment` is ignored, except when commentary is displayed.

Here's an example to illustrate how `'next` and `'alts` work:

```
#lang slideshow

(slide
 #:title "Example"
 (item "First step")
 'next
 (item "Second step")
 'next
 'alts
 (list (list (item "Tentative third step")
             'next
             (item "This isn't working... back up")))
       (list (item "Third step that works"))))
'next
(item "Fourth step"))
```

1.3 Display Size and Fonts

Slideshow is configured for generating slides in 1024 by 768 pixel format. When the current display has a different size as Slideshow is started, the Slideshow display still occupies the entire screen, and pictures are scaled just before they are displayed. Thus, one picture unit reliably corresponds to a “pixel” that occupies 1/1024 by 1/768 of the screen.

The `text` form for generating text pictures takes into account any expected scaling for the display when measuring text. (All Slideshow text functions, such as `t` and `item` are built on `text`.) In particular, scaling the picture causes a different font size to be used for drawing the slide—rather than bitmap-scaling the original font—and changing the font size by a factor of k does not necessarily scale all text dimensions equally by a factor of k —because, for most devices, each character must have integer dimensions. Nevertheless, especially if you use the `current-expected-text-scale` parameter, Slideshow is usually able to produce good results when the slide is scaled.

More generally, different font sets on different platforms can change the way a slide is rendered. For example, the `tt` font on one platform might be slightly wider than on another, causing different line breaks, and so on. Beware.

Beware also of using bitmaps in slides when the presentation screen is not 1024 by 768. In that case, consider using `size-in-pixels` (with the caveat that the resulting picture will take up different amounts of the slide on different displays).

1.4 Command-line Options

(require `slideshow/start`)

The `slideshow` executable instantiates the `slideshow/start` module, which inspects the command line as reported by `current-command-line-arguments` to get another module to require for the slide content. The `slideshow/start` module also initializes variables like `printing?` and `condense?` based on flags supplied on the command line.

Thus, if the above example is in "multi-step.rkt", then the command

```
slideshow multi-step.rkt
```

runs the slides.

If the module given to `slideshow` has a `slideshow` submodule, then `slideshow/start` requires the `slideshow` submodule after requiring the module. If the module has no `slideshow` but has a `main` submodule, then the `main` submodule is required.

The `slideshow` executable accepts a number of command-line flags. Use the `--help` flag to obtain a list of other flags.

1.5 Printing

The `-p` or `--print` command-line flag causes Slideshow to print slides instead of showing them on the screen using the current platform's printing system. The `-P` or `--ps` generates PostScript directly, while `-D` or `--pdf` generates PDF directly.

PS-to-PDF converters vary on how well they handle landscape mode. Here's a Ghostscript command that converts slides reliably (when you replace `"src.ps"` and `"dest.pdf"` with your file names):

```
gs -q -dAutoRotatePages=/None -dSAFER -dNOPAUSE -dBATCH -
sOutputFile=dest.pdf -sDEVICE=pdfwrite -c .setpdfwrite -c "<</Ori-
entation 3>> setpagedevice" -f src.ps
```


2 Making Pictures

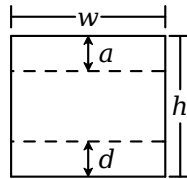
(require slideshow/pict)

The `slideshow/pict` layer provides core functions for constructing pictures, and it is independent of the slide viewer. This layer can be used, for example, to generate a picture as encapsulated PostScript for inclusion into a larger document. The `slideshow/pict` layer is re-provided by the `slideshow` language.

2.1 Pict Datatype

A *pict* is a `pict` structure representing an image. Some functions, such as `hline`, create new simple pict. Other functions, such as `ht-append`, build new pict out of existing pict. In the latter case, the embedded pict 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 pict 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 pict through arbitrarily complex computations on the size and shape of existing pict. 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`, and `'pdf-bytes`.

```

(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., its 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 sxy syx)
             #:extra-constructor-name make-child)
pict : pict?
dx : real?
dy : real?
sx : real?
sy : real?
sxy : real?
syx : 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 pict's to create a new one.

2.2 Basic Pict Constructors

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

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` and `h` arguments determine the width and height of the resulting pict's bounding box. In the three-argument case, the descent is 0 and the ascent is `h` for the bounding box; in the five-argument case, `a` and `d` are used as the bounding box's ascent and descent.

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))
```



```
(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 pics 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 as `'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 'italic style)`
- `(cons 'subscript style)`
- `(cons 'superscript style)`
- `(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, `colorizes` 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)
_____
> (vline 5 40 #:segment 5)
|
|
|
|
|

```

```

(frame pict
  [#:segment seg-length
   #:color color
   #:line-width width]) → pict?
pict : pict?
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, the 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) → pict?

```

```

w : real?
h : real?
(circle diameter) → pict?
  diameter : real?
(filled-ellipse w
                 h
                 [#:draw-border? draw-border?]) → pict?

w : real?
h : real?
draw-border? : any/c = #t
(disk diameter [#:draw-border? draw-border?]) → pict?
  diameter : real?
  draw-border? : any/c = #t





```

Unfilled and filled ellipses.

If `draw-border?` is `#f`, then the pen is set to be transparent before drawing the ellipse.

Examples:

```

> (ellipse 40 30)

> (circle 30)

> (filled-ellipse 30 40)

> (disk 30)


```

```

(rectangle w h) → pict?
  w : real?
  h : real?
(filled-rectangle w
                  h
                  [#:draw-border? draw-border?]) → pict?

w : real?
h : real?
draw-border? : any/c = #t

```

Unfilled and filled rectangles.

If `draw-border?` is `#f`, then the pen is set to be transparent before drawing the rectangle.

Examples:

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



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



```
(rounded-rectangle w
                   h
                   [corner-radius
                    #:angle angle]) → pict?

w : real?
h : real?
corner-radius : real? = -0.25
angle : real? = 0

(filled-rounded-rectangle w
                           h
                           [corner-radius
                            #:angle angle
                            #:draw-border? draw-border?]) → pict?

w : real?
h : real?
corner-radius : real? = -0.25
angle : real? = 0
draw-border? : any/c = #t
```

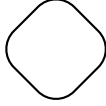
Unfilled and filled rectangles with rounded corners. The `corner-radius` is used to determine how much rounding occurs in the corners. If it is a positive number, then it determines the radius of a circle touching the edges in each corner, and the rounding of the rectangle follow the edge of those circles. 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 `width` and `height`.

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.

Examples:


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



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



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

A pict that display 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 word “bitmap failed”.

```
(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?]) → pict?

pict : pict?
src : pict-path?
find-src : (pict? pict-path? . -> . (values real? real?))
dest : pict-path?
find-dest : (pict? 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)
  (one-of/c 'transparent 'solid 'xor 'hilite
   'dot 'long-dash 'short-dash 'dot-dash
   'xor-dot 'xor-long-dash 'xor-short-dash
   'xor-dot-dash)
style :
  = 'solid

```

```

    under? : any/c = #f
(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?
               #:solid? solid?]
               #:hide-arrowhead? any/c) → pict?

arrow-size : real?
pict : pict?
src : pict-path?
find-src : (pict? pict-path? . -> . (values real? real?))
dest : pict-path?
find-dest : (pict? 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)
        (one-of/c 'transparent 'solid 'xor 'hilite
        style :
                'dot 'long-dash 'short-dash 'dot-dash
                'xor-dot 'xor-long-dash 'xor-short-dash
                'xor-dot-dash)
        = 'solid
under? : any/c = #f
solid? : any/c = #t
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?
                 #:solid? solid?]
                 #:hide-arrowhead? any/c) → pict?

arrow-size : real?
pict : pict?
src : pict-path?
find-src : (pict? pict-path? . -> . (values real? real?))
dest : pict-path?
find-dest : (pict? 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)
        (one-of/c 'transparent 'solid 'xor 'hilite
                  'dot 'long-dash 'short-dash 'dot-dash
                  'xor-dot 'xor-long-dash 'xor-short-dash
                  'xor-dot-dash)
style :
under? : any/c = #f
solid? : any/c = #t
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):

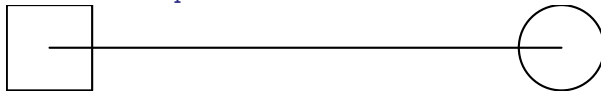
- The *start-angle* and *end-angle* arguments specify the direction of curve at its start and end positions; if either is *#f*, it defaults to the angle of a straight line from the start position to end position.
- The *start-pull* and *end-pull* arguments specify a kind of momentum for the starting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

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")
```



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



`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.

2.3 Pict Combiners

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

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"))

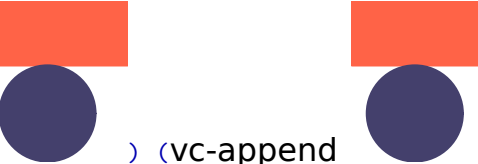


(define pict-a (colorize (filled-rectangle 60 30) "tomato"))




(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)






```






      
  
,((vl-append  ) (vc-append  ))
  

      
  
(vr-append  ) (ht-append  ))
```

> (drop picts 4)

```

      
  
,((htl-append   ) (hc-append   ))
  

      
  
(hbl-append   ) (hb-append   ))
```

```
(lt-superimpose pict ...) → pict?
  pict : pict?
(ltl-superimpose pict ...) → pict?
  pict : pict?
(lc-superimpose pict ...) → pict?
```

```

  pict : pict?
(lbl-superimpose pict ...) → pict?
  pict : pict?
(lb-superimpose pict ...) → pict?
  pict : pict?
(ct-superimpose pict ...) → pict?
  pict : pict?
(ctl-superimpose pict ...) → pict?
  pict : pict?
(cc-superimpose pict ...) → pict?
  pict : pict?
(cbl-superimpose pict ...) → pict?
  pict : pict?
(cb-superimpose pict ...) → pict?
  pict : pict?
(rt-superimpose pict ...) → pict?
  pict : pict?
(rtl-superimpose pict ...) → pict?
  pict : pict?
(rc-superimpose pict ...) → pict?
  pict : pict?
(rbl-superimpose pict ...) → pict?
  pict : pict?
(rb-superimpose pict ...) → pict?
  pict : pict?

```

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)
'((lt-superimpose  ) (ltl-superimpose  ))
(lc-superimpose  ))
> (take (drop picts 3) 3)
'((lbl-superimpose  ) (lb-superimpose  ))
(ct-superimpose  ))
> (take (drop picts 6) 3)
'((ctl-superimpose  ) (cc-superimpose  ))
(cbl-superimpose  ))
> (take (drop picts 9) 3)

```

```

'((cb-superimpose  ) (rt-superimpose  )
  (rtl-superimpose  ))
> (take (drop picts 12) 3)
'((rc-superimpose  ) (rbl-superimpose  )
  (rb-superimpose  ))

```

```

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

```

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`.

```

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

```

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

```

(table ncols
      picts
      col-aligns
      row-aligns
      col-seps
      row-seps) → pict?
ncols : exact-positive-integer?
picts : (non-empty-listof pict?)
col-aligns : (list*of (pict? pict? -> pict?))
row-aligns : (list*of (pict? pict? -> pict?))
col-seps : (list*of real?)
row-seps : (list*of real?)

```

Creates a table given a list of pict. 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.

2.4 Pict Drawing Adjusters

```

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

```

Scales a pict drawing, as well as its bounding box. The drawing is scaled by adjusting the destination `dc<%>`’s scale while drawing the original *pict*.

```

(scale-to-fit pict size-pict) → pict?
  pict : pict?
  size-pict : pict?
(scale-to-fit pict width height) → pict?

```

```

pict : pict?
width : real?
height : real?

```

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).

The aspect ratio of the *pict* is preserved, so the resulting *pict* will have either the width or the height of the *size-pict* (or *width* by *height* box), but not necessarily both.

```

(rotate pict theta) → pict?
pict : pict?
theta : real?

```

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

The bounding box of the resulting *pict* is the box that 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.

```

(ghost pict) → pict?
pict : pict?

```

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

```

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

```

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").

```

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

```

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

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

```
(cellophane pict opacity) → pict?
  pict : pict?
  opacity : (real-in 0 1)
```

Makes the given *pict* semi-transparent, where an opacity of 0 is fully transparent, and an opacity of 1 is fully opaque. See `set-alpha` for information about the contexts and cases when semi-transparent drawing works.

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

Clips a *pict*'s drawing to its bounding box.

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

```
(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*.

```
(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.

2.5 Bounding Box Adjusters

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

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

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

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

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

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

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

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

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

```
(refocus pict sub-pict) → pict?  
pict : pict?  
sub-pict : pict?
```

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*).

```
(panorama pict) → pict?  
pict : pict?
```

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

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

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*.

2.6 Pict Finders

```
(lt-find pict find) → real? real?  
pict : pict?  
find : pict-path?  
(l1-find pict find) → real? real?
```

```

    pict : pict?
    find : pict-path?
(lc-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(lbl-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(lb-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(ct-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(ctl-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(cc-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(cbl-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(cb-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(rt-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(rtl-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(rc-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(rbl-find pict find) → real? real?
    pict : pict?
    find : pict-path?
(rb-find pict find) → real? real?
    pict : pict?
    find : pict-path?

```

Locates a pict designated by *find* is 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 second element, and so on.

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

Returns #t if *v* is a *pict* or a non-empty list of *picts*.

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

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.

2.7 More Pict Constructors

2.7.1 Dingbats

```
(cloud w h [color]) → pict?  
w : real?  
h : real?  
color : (or/c string? (is-a?/c color%)) = "gray"
```

Creates a fluffy cloud.

Examples:

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



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



```
(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)
```



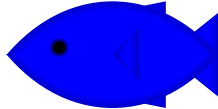
```
(standard-fish w
               h
               [#:direction direction
               #:color color
               #:eye-color eye-color
               #:open-mouth open-mouth]) → pict?
w : real?
h : real?
direction : (or/c 'left 'right) = 'left
color : (or/c string? (is-a?/c color%)) = "blue"
eye-color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

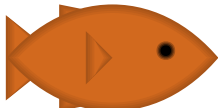
The `open-mouth` argument can be either #f (mouth closed), #t (mouth fully open), or a number: 0.0 is closed, 1.0 is fully open, and numbers in between are partially open.

Examples:

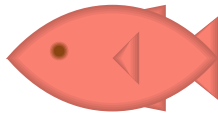
```
> (standard-fish 100 50)
```



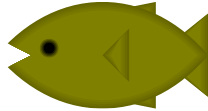
```
> (standard-fish 100 50 #:direction 'right #:color "chocolate")
```



```
> (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]) → pict?
size : real?
pumpkin-color : (or/c string? (is-a?/c color%)) = "orange"
face-color : (or/c string? (is-a?/c color%)) = "black"
```

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")
```



```
(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
```

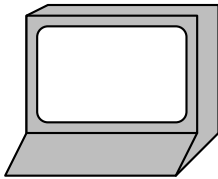
Produces a picture of ancient desktop computer. The *scale* argument scales the size relative to the base size of 120 by 115.

The *style* can include any of the following:

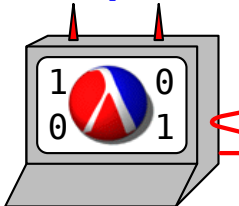
- `'plt` — include a Racket logo on the machine's screen
- `'binary` — put 1s and 0s on the machine's screen
- `'devil` — like `'binary`, and also give the machine horns and a tail

Examples:

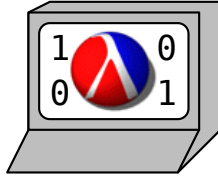
```
> (desktop-machine 1)
```



```
> (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 : non-exact-negative-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 : positive-real? = 40
bottom-circle-diameter : positive-real? = 80
stem-height : positive-real? = 180
mercury-inset : positive-real? = 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.

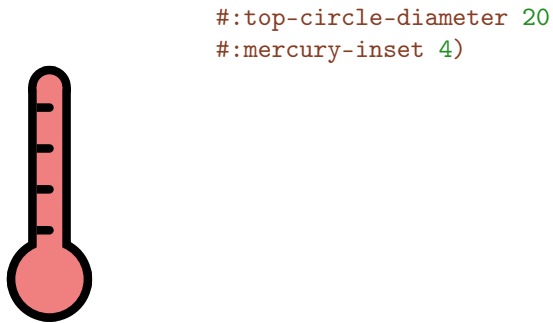
The mercury is drawn the same way, but by creating the three components inset from the versions that draw the boundary of the thermometer. This inset is controlled by the *mercury-inset* argument.

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
```



2.7.2 Balloon Annotations

```
(require slideshow/balloon)
```

The `slideshow/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%)) = 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%)) = 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?))
```

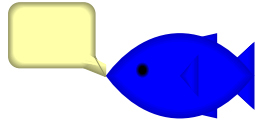
Superimposes the pict in `balloon` onto `base` to produce a new pict. The balloon is positioned so that its spike points to the location specified by either `x` and `y` (numbers) or at the position determined by combining `base` and `at-pict` with `find`. The `find` function uses its arguments like `lt-find`.

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

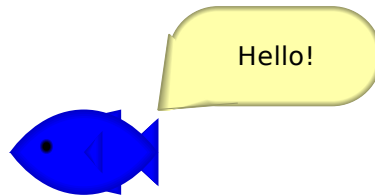
Examples:

```
> (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%)) = 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.

```

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

```

A parameter that determines whether balloons are drawn with 3-D shading.

2.7.3 Face

```

(require slideshow/face)

```

The `slideshow/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.

2.7.4 Flash

```
(require slideshow/flash)

(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))
```

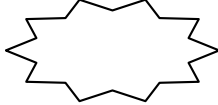


```
(outline-flash width
               height
               [n-points
               spike-fraction
               rotation]) → pict?
width : real?
height : real?
n-points : exact-positive-integer? = 10
spike-fraction : (real-in 0 1) = 0.25
rotation : real? = 0
```

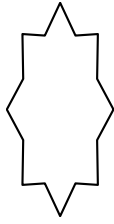
Like *filled-flash*, but drawing only the outline.

Examples:

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



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



2.8 Miscellaneous

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

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

```
(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.

```
(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.

2.9 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 pict's 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.

```
(draw-pict pict dc x y) → void?
  pict : pict?
  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) → (is-a?/c bitmap%)
  pict : pict?
```

Returns a `bitmap%` with an alpha channel, no larger than `pict`, with `pict` drawn on it in the top-left corner $(0, 0)$.

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

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?
  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)
```

```

      (listof (or/c 'no-resize-border 'no-caption
                  'no-system-menu 'hide-menu-bar
                  'toolbar-button 'float 'metal))
    frame-style :

```

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 real? real?)
(current-expected-text-scale scales) → void?
scales : (list 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.

2.10 Conversion to Picts

```
(require slideshow/pict-convert)
```

The `slideshow/pict-convert` library defines a protocol for values to convert themselves to pict. The protocol is used by DrRacket's interactions window, for example, to render values that it prints

```
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 struct are convertible to pict, but others are not).

```

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

```


Returns `#t` if `v` supports the conversion protocol (by being a struct with the `prop:pict-convertible` property) and `#f` otherwise.

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

Requests a data conversion from `v` to a pict.

3 Making Slides

(require slideshow/base)

The `slideshow/base` module, which is re-provided by `slideshow`, provides the functions for creating slides.

3.1 Primary Slide Functions

```
(slide [#:title title
        #:name name
        #:layout layout
        #:gap-size sep-gap-size
        #:inset inset
        #:timeout secs
        #:condense? condense?]
       element ...) → void?
title : (or/c #f string? pict?) = #f
name : (or/c #f string?) = title
layout : (or/c 'auto 'center 'top 'tall) = 'auto
sep-gap-size : real? = (current-gap-size)
inset : slide-inset? = (make-slide-inset 0 0 0 0)
secs : (or/c #f real?) = #f
condense? : any/c = (and timeout #t)
          (flat-rec-contract elem/c
            (or/c pict?
              'next 'next! 'alts 'alts~ 'nothing
              comment?
              (listof (listof elem/c))))
element :
```

Creates and registers a slide. See §1.2 “Staging Slides” for information about `elements`. Multiple `element` picts are separated by `sep-gap-size` vertical space.

When this function is first called in non-printing mode, then the viewer window is opened. Furthermore, each call to the function `yields`, so that the viewer window can be refreshed, and so the user can step through slides.

If `title` is not `#f`, then a title is shown for the slide. The `name` is used in the slide-navigation dialog, and it defaults to `title`.

If `layout` is `'top`, then the content is top-aligned, with `(* 2 sep-gap-size)` space between the title and the content. The `'tall` layout is similar, but with only `sep-gap-size` space. The `'center` mode centers the content (ignoring space consumed by the title). The `'auto` mode is like `'center`, except when `title` is non-`#f` and when the space between

the title and content would be less than `(* 2 sep-gap-size)`, in which case it behaves like `'top`.

The `inset` argument supplies an inset that makes the slide-viewing window smaller when showing the slide. See `make-slide-inset` for more information.

If `secs` argument for `#:timeout` is not `#f`, then the viewer automatically advances from this slide to the next after `secs` seconds, and manual advancing skips this slide.

If `condense?` is true, then in condense mode (as specified by the `-c` command-line flag), the slide is not created and registered.

```
(t str) → pict?  
str : string?
```

The normal way to make plain text. Returns `(text str (current-main-font) (current-font-size))`.

```
(it str) → pict?  
str : string?
```

The normal way to make italic text. Returns `(text str (cons 'italic (current-main-font)) (current-font-size))`.

```
(bt str) → pict?  
str : string?
```

The normal way to make bold text. Returns `(text str (cons 'bold (current-main-font)) (current-font-size))`.

```
(bit str) → pict?  
str : string?
```

Bold-italic text. Returns `(text str (list* 'bold 'italic (current-main-font)) (current-font-size))`.

```
(tt str) → pict?  
str : string?
```

The normal way to make monospaced text. Returns `(text str '(bold . modern) (current-font-size))`.

```
(rt str) → pict?  
str : string?
```

The normal way to make serif text. Returns `(text str 'roman (current-font-size))`.

```
(titlelet str) → pict?
  str : string?
```

Creates title text. Returns ((current-titlelet) str).

```
(para [#:width width
      #:align align
      #:fill? fill?
      #:decode? decode?]
      element ...) → pict?
width : real? = (current-para-width)
align  : (or/c 'left 'center 'right) = 'left
fill?  : any/c = #t
decode? : any/c = #t
element : (flat-rec-contract elem/c
          (or/c string? pict? (listof elem/c)))
```

Generates a paragraph pict that is no wider than *width* units, and that is exactly *width* units if *fill?* is true. If *fill?* is #f, then the result pict is as wide as the widest line.

Each list within *elements* is spliced into the sequence of string and pict elements. If *decode?* is true, then strings among the *elements* are decoded by performing the following substitutions: `---` → `--`, `--` → `=`, `‘‘` → `‘`, `’’` → `’`, `?’` → `?`. In addition, to better work with `at-exp` notation, if an *element* is `"\n"`, then it is dropped along with any spaces at the start of the next element.

Strings are split at spaces for word-wrapping to fit the page, and a space is added between elements. If a string element starts with one of the following punctuation marks (after decoding), however, no space is added before the string:

```
= ? , . : ; ? ! ) ' ' ?
```

The *align* argument specifies how to align lines within the paragraph.

See the spacing between lines is determined by the `current-line-sep` parameter.

```
(item [#:width width
      #:gap-size sep-gap-size
      #:bullet blt
      #:align align
      #:fill? fill?
      #:decode? decode?]
      element ...) → pict?
width : real? = (current-para-width)
sep-gap-size : real? = (current-gap-size)
blt : pict? = (scale bullet (/ sep-gap-size gap-size))
```

```

align : (or/c 'left 'center 'right) = 'left
fill? : any/c = #t
decode? : any/c = #t
element : (flat-rec-contract elem/c
           (or/c string? pict? (listof elem/c)))

```

Like `para`, but with `blt` followed by `(/ sep-gap-size 2)` space appended horizontally to the resulting paragraph, aligned with the top line. The paragraph width of `blt` plus `(/ sep-gap-size 2)` is subtracted from the maximum width of the paragraph.

```

(subitem [#:width width
         #:gap-size sep-gap-size
         #:bullet blt
         #:align align
         #:fill? fill?
         #:decode? decode?]
        element ...) → pict?
width : real? = (current-para-width)
sep-gap-size : real? = (current-gap-size)
blt : pict? = (scale o-bullet (/ sep-gap-size gap-size))
align : (or/c 'left 'center 'right) = 'left
fill? : any/c = #t
decode? : any/c = #t
element : (flat-rec-contract elem/c
           (or/c string? pict? (listof elem/c)))

```

Like `item`, but an additional `(* 2 sep-gap-size)` is subtracted from the paragraph width and added as space to the left of the pict. Also, `o-bullet` is the default bullet, instead of `bullet`.

```

(clickback pict thunk) → pict?
pict : pict?
thunk : (-> any)

```

Creates a pict that embeds the given one, and is the same size as the given pict, but that when clicked during a presentation calls `thunk`.

```

(interactive pict proc) → pict?
pict : pict?
proc : (frame% . -> . (-> any))

```

Creates a pict that embeds the given one, but that creates a floating frame at the pict's location on the screen during a presentation. After the floating frame is created (and before it is shown), `proc` is applied to the frame. The result from `proc` must be a procedure that is called when the window is removed (because the slide changes, for example).

```
(size-in-pixels pict) → pict?  
  pict : pict?
```

Scales *pict* so that it is displayed on the screen as (*pict-width pict*) pixels wide and (*pict-height pict*) pixels tall. The result is *pict* when using a 1024 by 768 display.

```
(make-outline name title subitems ...) → (symbol? . -> . void?)  
  name : (or/c symbol? (listof symbol?))  
  title : (or/c string? pict?)  
  subitems : (or/c #f null?  
             (symbol? . -> . pict?))
```

Returns a function that takes a symbol and generates an outline slide.

The ... above applies to all three arguments together. Each trio of arguments defines a section for the outline:

- The section *name* is either a symbol or a list of symbols. When the outline function is called later to make an outline, the given symbol is compared to the section's symbol(s), and the section is marked as current if the symbol matches.
- The *title* is used as the displayed name of the section.
- The *subitems* are displayed when the section is active. It can be *#f* or *null* (for historical reasons) if no subitems are to be displayed. Otherwise, it should be a function that takes a symbol (the same one passed to the outline maker) and produces a pict.

```
(comment text ...) → comment?  
  text : (or/c string? pict?)
```

Combines strings and picts to be used as a slide element for (usually hidden) commentary. Use the result as an argument to *slide*.

```
(comment? v) → boolean?  
  v : any/c
```

Returns *#t* if *v* is a comment produced by *comment*.

3.2 Slide Registration

```
(slide? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` is a slide produced by `most-recent-slide` or `retract-most-recent-slide`.

```
(most-recent-slide) → slide?
```

Returns a slide structure that be supplied `re-slide` to make a copy of the slide or `slide->pict` to re-extract the entire slide as a pict.

```
(retract-most-recent-slide) → slide?
```

Cancels the most recently created slide, and also returns a slide structure that be supplied to `re-slide` to restore the slide (usually in a later position).

```
(re-slide slide [pict]) → void?  
  slide : slide?  
  pict : pict? = (blank)
```

Re-inserts a slide, `lt-superimpose`ing the given additional `pict`.

```
(slide->pict slide) → pict?  
  slide : slide?
```

Converts a complete slide to a `pict`. The bounding box of the result corresponds to the slide within its margins.

3.3 Viewer Control

```
(start-at-recent-slide) → void?
```

Sets the starting slide for the talk to the most recently created slide. If this function is used multiple times, the last use overrides the earlier uses.

```
(enable-click-advance! on?) → void?  
  on? : any/c
```

Enables or disables slide advance as a result of a mouse click.

```
(set-use-background-frame! on?) → void?  
  on? : any/c
```

Enables or disables the creation of a background frame, which is typically useful only when `make-slide-inset` is used are active. The last enable/disable before the first slide registration takes effect once and for all.

```
(set-page-numbers-visible! on?) → void?  
  on? : any/c
```

Determines whether slide numbers are initially visible in the viewer.

```
(current-page-number-font) → (is-a?/c font%)  
(current-page-number-font font) → void?  
  font : (is-a?/c font%)
```

Parameter that determines the font used to draw the page number (if visible).

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

Parameter that determines the color used to draw the page number (if visible).

```
(current-page-number-adjust) → (-> number? string? string?)  
(current-page-number-adjust proc) → void?  
  proc : (-> number? string? string?)
```

Parameter that controls the precise text that appears to indicate the page numbers (if visible). The input to the function is the default string and the slide number, and the result is what is drawn in the bottom right corner. The default parameter value just returns its first argument.

3.4 Constants and Layout Variables

```
gap-size : 24
```

A width commonly used for layout.

```
(current-gap-size) → real?  
(current-gap-size sep-gap-size) → void?  
  sep-gap-size : real?
```

A parameter whose value is a width used for the separation between items by `slide`, the size and spacing of a bullet for `item`, the space between a slide title and content in `'tall` mode, etc. The default value is `gap-size`.

```
bullet : pict?
```


A filled bullet used by default by `item`.

It is either `(t "●")`, if that character is available in the font that `t` uses, or it uses an implementation similar to `o-bullet`, but not hollow (using `disk`, not `circle`).

`o-bullet` : pict?

A hollow bullet used by default by `subitem`.

It's implementation is:

```
(baseless
  (cc-superimpose
    (circle (/ gap-size 2))
    (blank 0 gap-size)))
```

`client-w`

Produces the width of the display area, minus margins. The result of the form changes if the margin is adjusted via `set-margin!`.

`client-h`

Produces the height of the display area, minus margins, but including the title area). The result of the form changes if the margin is adjusted via `set-margin!`.

`full-page`

Produces an empty pict that is the same size as the client area, which is like `(blank client-w client-h)`.

`titleless-page`

Produces an empty pict that is the same size as the client area minus the title area in 'top layout mode, which is like `(blank client-w (- client-h title-h (* 2 gap-size)))`.

`margin`

Produces a number that corresponds to the current margin, which surrounds every side of the slide. The client area for a slide corresponds to the display area (which is always 1024 by 768) minus this margin on each side. The default margin is 20.

The margin can be adjusted via `set-margin!`.

`title-h`

Produces a number that corresponds to the height of a title created by `titlet`.

If `titlet` is changed via the `current-titlet` parameter, the title height should be updated via `set-title-h!`.

`printing?` : `boolean?`

The value is `#t` if slides are being generated for printed output, `#f` for normal on-screen display. Printing mode is normally triggered via the `--print` or `--ps` command-line flag.

`condense?` : `boolean?`

The value is `#t` if slides are being generated in condensed mode, `#f` for normal mode. Condensed mode is normally triggered via the `--condense` command-line flag.

3.5 Configuration

```
(current-font-size) → exact-nonnegative-integer?  
(current-font-size n) → void?  
  n : exact-nonnegative-integer?
```

Parameter that determines the font size used by `t`, `para`, etc. The default size is `32`.

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

Parameter that determines the font size used by `t`, `para`, etc. The default is platform-specific; possible initial values include `'swiss`, `"Verdana"`, and `"Gill Sans"`.

```
(current-line-sep) → exact-nonnegative-integer?  
(current-line-sep n) → void?  
  n : exact-nonnegative-integer?
```

Parameter that controls the amount of space used between lines by `para`, `item`, and `subitem`.

```
(current-para-width) → exact-nonnegative-integer?  
(current-para-width n) → void?  
  n : exact-nonnegative-integer?
```

Parameter that controls the width of a pict created by `para`, `item`, and `subitem`.

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

Parameter used by the default `current-titlet` to colorize the title. The default is "black".

```
(current-slide-assembler) → ((or/c string? #f)
                               exact-nonnegative-integer?
                               pict?)
                               . -> .
                               pict?)
(current-slide-assembler proc) → void?
  ((or/c string? #f)
   exact-nonnegative-integer?
   proc : pict?
   . -> .
   pict?)
```

Parameter whose value is a function for assembling slide content into a single pict; the assembling function takes a string for the title (or `#f`), a separation for the title (if any) and pict, and a pict for the slide content (not counting the title).

The result is of the assembler is `ct-superimposed` with the client area, but the result pict might draw outside the client region to paint the screen margins, too.

The default assembler uses `titlet` to turn a title string (if any) to a pict. See also `current-titlet` and `set-title-h!`.

The slide assembler is *not* responsible for adding page numbers to the slide; that job belongs to the viewer. See also `current-page-number-font`, `current-page-number-color`, and `set-page-numbers-visible!`.

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

Parameter to configure `titlet`. The default is

```
(lambda (s)
  (colorize (text s (current-main-font) 40)
            (current-title-color)))
```

If this parameter is changed such that the result is a different height, then `set-title-h!` should be called to update the value produced by `title-h`, `titleless-page`, etc.

```
(set-margin! amt) → void?
  amt : real?
```

Changes the margin that surrounds the client area. See also `margin`.

```
(set-title-h! amt) → void?
  amt : real?
```

Changes the expected height of a title, which adjusts `title-h`, `client-h`, `full-page`, and `titleless-page`.

```
(make-slide-inset left-inset
                  top-inset
                  right-inset
                  bottom-inset) → slide-inset?
  left-inset : exact-nonnegative-integer?
  top-inset  : exact-nonnegative-integer?
  right-inset : exact-nonnegative-integer?
  bottom-inset : exact-nonnegative-integer?
```

Creates a slide inset, which describes a number of pixels to inset the viewer for a slide on each side.

```
(slide-inset? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a slide inset created by `make-slide-inset`, `#f` otherwise.

3.6 Pict-Staging Helper

```
(require slideshow/step)
```

The `slideshow/step` library provides syntax for breaking a complex slide into steps that are more complex than can be handled with `'next` and `'alts` in a `slide` sequence.

```
(with-steps (id ...) body ...)
```

Evaluates the `bodys` once for each `id`, skipping an `id` if its name ends with `~` and `condense?` is true. The results of the last `body` for each iteration are collected into a list, which is the result of the `with-steps` form.

Within the `bodys`, several keywords are bound non-hygenically (using the first `body`'s lexical context):

- `(only? id)` — returns `#t` during the `id` step (i.e., during the evaluation of the `bodys` for `id`), `#f` otherwise.
- `(vonly id)` — returns the identity function during the `id` step, `ghost` otherwise.
- `(only id then-expr)` returns the result of `then-expr` during the `id` step, `values` otherwise.
- `(only id then-expr else-expr)` returns the result of `then-expr` during the `id` step, the result of `else-expr` otherwise.
- `(before? id)` — returns `#t` before the `id` step, `#f` starting for the `id` and afterward.
- `(vbefore id)`, `(before id then-expr)`, or `(before id then-expr else-expr)` — analogous to `vonly` and `only`.
- `(after? id)` — returns `#t` after the `id` step, `#f` through the `id` step.
- `(vafter id)`, `(after id then-expr)`, or `(after id then-expr else-expr)` — analogous to `vonly` and `only`.
- `(between? a-id b-id)` — returns `#t` starting from the `a-id` step through the `b-id` step, `#f` otherwise.
- `(vbetween a-id b-id)`, `(between a-id b-id then-expr)`, or `(between a-id b-id then-expr else-expr)` — analogous to `vonly` and `only`.
- `(between-excel? a-id b-id)` — returns `#t` starting from the `a-id` step through steps before the `b-id` step, `#f` for the `b-id` step and afterward.
- `(vbetween-excl a-id b-id)`, `(between-excl a-id b-id then-expr)`, or `(between-excl a-id b-id then-expr else-expr)` — analogous to `vonly` and `only`.

```
(with-steps~ (id ...) body ...)
```

Like `with-steps`, but when `condense?` is true, then `expr` is evaluated only for the last `id` (independent of whether the name for the last `id` name ends in `~`).

3.7 Slides to Picts

```
(require slideshow/slides-to-picts)
```

```
(get-slides-as-picts path
                    width
                    height
                    condense?
                    [stop-after]) → (listof pict?)
```

```
path : path-string?  
width : real?  
height : real?  
condense? : any/c  
stop-after : (or/c #f exact-nonnegative-integer?) = #f
```

Executes the Slideshow program indicated by *path* in a fresh namespace, and returns a list of pict for the slides. Each pict has the given *width* and *height*, and *condense?* determines whether the Slideshow program is executed in condense mode.

If *stop-after* is not *#f*, then the list is truncated after *stop-after* slides are converted to pict.

4 Typesetting Racket Code

```
(require slideshow/code-pict)
(require slideshow/code)
```

The `slideshow/code-pict` library provides utilities for typesetting Racket code as a `pict`. The `slideshow/code` library initializes `get-current-code-font-size` to `current-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. Also beware that when a source file containing a `syntax` or `quote-syntax` form is compiled, source location information is omitted from the compiled syntax object.

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 block, with each `s` on its own line, where each `s` must be a string or a `pict`.
- `(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 under a quote is colored 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 superscripts 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 `datums`:

```
(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 un-syntax—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 intected 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 *datums* in a begin. In this case, *_s* are removed from the *datums*, 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)))`.

5 Animations

```
(require slideshow/play)
```

The `slideshow/play` module provides tools for generating animations as multiple, automatically advanced slides.

Many of the tools are based on a function that takes a number between `0.0` and `1.0` inclusive and produces a `pict`. The `pict` produced for the input `0.0` is the starting image of the animation, and the `pict` produced for `1.0` is the ending image, while intermediate values produced intermediate images. For example,

```
(lambda (n)
  (cellophane (t "Hello") n))
```

corresponds to an animation that fades in the word “Hello.”

5.1 Generating Animated Slides

```
(play gen
  [#:steps steps
   #:delay delay-secs
   #:skip-first? skip-first?
   #:title title
   #:name name
   #:layout layout]) → void?
gen : ((real-in 0.0 1.0) . -> . pict?)
steps : exact-positive-integer? = 10
delay-secs : real? = 0.05
skip-first? : any/c = #f
title : (or/c string? pict? #f
         ((real-in 0.0 1.0) . -> . (or/c string? pict? #f)))
       = #f
name : (or/c string? #f
        ((real-in 0.0 1.0) . -> . (or/c string? #f)))
      = title
layout : (or/c 'auto 'center 'top 'tall) = 'auto
```

Generates `steps+1` slides by calling `gen` on equally-spaced values from `0.0` (inclusive) to `1.0` (exclusive). Except for the first of the slides, each slide has a timeout of `delay-secs`, so that the next slide appears automatically.

Normally, `play` is called via `play-n`, which effectively calls `gen` on `1.0` without a timeout

to complete the animation and stop the auto-advance of slides. The `play-n` function also manages with multi-step animations.

If `skip-first?` is `#f`, then one fewer slide is generated, because `gen` is not called on `0.0`.

The `title`, `name`, and `layout` arguments are passed on to `slide`, at least when `title` and/or `name` are not functions. When `title` or `name` is a function, the function is applied to the value used to produce the slide content, and the resulting title or name is passed on to `slide`.

In condensed mode (i.e., when `condense?` is `#t`), any slide that would be registered with a timeout is instead skipped.

```
(play-n gen*
  [#:steps steps
   #:delay delay-secs
   #:skip-first? skip-first?
   #:skip-last? skip-last?
   #:title title
   #:name name
   #:layout layout]) → void?
gen* : ((() (listof (real-in 0.0 1.0)) . ->* . pict?)
steps : (or/c exact-positive-integer?
         (improper-listof exact-positive-integer?)) = 10
delay-secs : real? = 0.05
skip-first? : any/c = #f
skip-last? : any/c = #f
title : (or/c string? pict? #f
         ((real-in 0.0 1.0) . -> . (or/c string? pict? #f)))
       = #f
name : (or/c string? #f
        ((real-in 0.0 1.0) . -> . (or/c string? #f)))
      = title
layout : (or/c 'auto 'center 'top 'tall) = 'auto
```

Generates a sequence of slides by calling `gen*` with, for each of its arguments, numbers from `0.0` to `1.0`. If `gen*` accepts n arguments, then result is a sequence of animations with a pause (i.e., not auto-advanced) between each of n segments.

If `gen*` accepts a single argument, then `play-n` is like `play`, except that `gen*` is also called with `1.0` to generate a slide with no timeout. If `gen*` accepts multiple arguments, then slides are generated by calling `gen*` with the first argument varying from `0.0` to `1.0` while all other arguments are `0.0`. Then, the first argument is held at `1.0` while the second argument varies from `0.0` to `1.0`, and so on.

For example,


```
(play-n
  (lambda (n1 n2)
    (cellophane (t "Hello")
      (* n1 (- 1.0 n2)))))
```

generates an animation to fade in the word “Hello,” and then pauses for a manual advance, and then fades “Hello” back out.

If *skip-first?* is #t, then the very first slide of the sequence is skipped. Similarly, if *skip-last?* is #t, then the last slide of the sequence is skipped.

The *steps* argument controls how many steps happen in each phase on the animation. If it is a number, then that number is used for each phase. If it is a pair of two numbers, then the first number is used for the first phase, and the second number is used for the rest of the phases. Similarly, if it is (cons num_1 (cons num_2 num_3)), num_1 and num_2 are used for the first two phases and num_3 is used for the rest.

The *delay-msecs*, *title*, *name*, and *layout* arguments are passed on to `play` for each of the *n* segments of animation.

```
(animate-slide element ...)
→ (() (listof (real-in 0.0 1.0)) . ->* . pict?)
      (flat-rec-contract elem/c
        element : (or/c pict? 'next 'alts
          (listof (listof elem/c))))
```

Accepts slide content similar to `slide` with *'next* and *'alts* and produces a procedure suitable for use with `play-n`. The result is similar to using `slide`, but with fades for *'next* and *'alts* transitions (to better fit the style, perhaps, of surrounding animations).

5.2 Animation Helpers

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

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.

```
(fade-around-pict n p1 make-p2) → pict?
  n : (real-in 0.0 1.0)
  p1 : pict?
  make-p2 : (pict? . -> . pict?)
```

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,

```
(lambda (n)
  (fade-around-pict n
    (code x)
    (lambda (g) (code (+ #,x 1)))))
```

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

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

Pins *p* onto *base*, sliding from *p-from* to *p-to* (which are pict 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* pict are typically *laundered ghosts* of *p* within *base*, but they can be any pict within *base*.

5.3 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.

5.4 Stretching and Squashing Time

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

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

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.

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.

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.

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

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

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

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

Bibliography

- [Findler06] Robert Bruce Findler and Matthew Flatt, “Slideshow: Functional Presentations,” *Journal of Functional Programming*, 16(4-5), pp. 583–619, 2006. <http://www.cs.utah.edu/plt/publications/jfp05-ff.pdf>

Index

[angel-wing](#), 35
[animate-slide](#), 73
Animation Helpers, 73
Animations, 71
[arrow](#), 17
[arrowhead](#), 17
[balloon](#), 40
Balloon Annotations, 38
[balloon-color](#), 41
[balloon-enable-3d](#), 41
[balloon-pict](#), 40
[balloon-point-x](#), 40
[balloon-point-y](#), 41
[balloon?](#), 40
[baseless](#), 31
Basic Pict Constructors, 11
[bit](#), 51
[bitmap](#), 17
[bitmap-draft-mode](#), 22
[black-and-white](#), 30
[blank](#), 12
bounding box, 9
Bounding Box Adjusters, 30
[bt](#), 51
[bullet](#), 56
[cb-find](#), 32
[cb-superimpose](#), 24
[cbl-find](#), 32
[cbl-superimpose](#), 24
[cc-find](#), 32
[cc-superimpose](#), 24
[cellophane](#), 29
[child](#), 10
[child-dx](#), 10
[child-dy](#), 10
[child-pict](#), 10
[child-sx](#), 10
[child-sxy](#), 10
[child-sy](#), 10
[child-syx](#), 10
[child?](#), 10
[circle](#), 15
[clickback](#), 53
[client-h](#), 57
[client-w](#), 57
[clip](#), 29
[clip-descent](#), 30
[cloud](#), 33
[code](#), 64
code transformers, 68
[code-align](#), 66
[code-colorize-enabled](#), 67
[code-colorize-quote-enabled](#), 67
[code-italic-underscore-enabled](#), 67
[code-pict-bottom-line-pict](#), 69
[code-scripts-enabled](#), 67
[code-transformer?](#), 68
[code:blank](#), 63
[code:comment](#), 63
[code:contract](#), 64
[code:line](#), 63
[code:template](#), 64
[color-series](#), 46
[colorize](#), 29
Command-line Options, 7
[comment](#), 54
[comment-color](#), 69
[comment?](#), 54
[condense?](#), 58
Configuration, 58
Constants and Layout Variables, 56
Conversion to Picts, 48
Creating Slide Presentations, 4
[ct-find](#), 32
[ct-superimpose](#), 24
[ctl-find](#), 32
[ctl-superimpose](#), 24
[current-base-color](#), 66
[current-code-font](#), 64
[current-code-line-sep](#), 65
[current-code-tt](#), 64
[current-comment-color](#), 65

[current-const-color](#), 65
[current-const-list](#), 66
[current-expected-text-scale](#), 48
[current-font-size](#), 58
[current-gap-size](#), 56
[current-id-color](#), 65
[current-keyword-color](#), 65
[current-keyword-list](#), 66
[current-line-sep](#), 58
[current-literal-color](#), 65
[current-literal-list](#), 66
[current-main-font](#), 58
[current-page-number-adjust](#), 56
[current-page-number-color](#), 56
[current-page-number-font](#), 56
[current-para-width](#), 58
[current-reader-forms](#), 66
[current-slide-assembler](#), 59
[current-title-color](#), 59
[current-titlet](#), 59
[dc](#), 11
[dc-for-text-size](#), 47
[default-face-color](#), 41
[define-code](#), 67
[define-exec-code](#), 69
[define-exec-code/scale](#), 69
[desktop-machine](#), 36
[Dingbats](#), 33
[disk](#), 15
[Display Size and Fonts](#), 7
[draw-pict](#), 47
[drop-below-ascent](#), 30
[ellipse](#), 14
[enable-click-advance!](#), 55
[Face](#), 41
[face](#), 41
[face*](#), 43
[fade-around-pict](#), 74
[fade-pict](#), 73
[fast-edges](#), 75
[fast-end](#), 75
[fast-middle](#), 75
[fast-start](#), 75
[file-icon](#), 33
[filled-ellipse](#), 15
[filled-flash](#), 44
[filled-rectangle](#), 15
[filled-rounded-rectangle](#), 16
[Flash](#), 44
[frame](#), 14
[full-page](#), 57
[gap-size](#), 56
[Generating Animated Slides](#), 71
[get-current-code-font-size](#), 65
[get-slides-as-picts](#), 61
[ghost](#), 28
[hb-append](#), 22
[hbl-append](#), 22
[hc-append](#), 22
[hline](#), 14
[ht-append](#), 22
[html-append](#), 22
[hyperlinkize](#), 46
[id-color](#), 69
[inset](#), 30
[inset/clip](#), 29
[interactive](#), 53
[inverted point](#), 10
[it](#), 51
[item](#), 52
[jack-o-lantern](#), 35
[keyword-color](#), 69
[launder](#), 33
[lb-find](#), 32
[lb-superimpose](#), 24
[lbl-find](#), 32
[lbl-superimpose](#), 24
[lc-find](#), 32
[lc-superimpose](#), 23
[lift-above-baseline](#), 30
[linestyle](#), 28
[linewidth](#), 28
[literal-color](#), 69
[lt-find](#), 31

[lt-superimpose](#), 23
[ltl-find](#), 31
[ltl-superimpose](#), 23
[make-balloon](#), 40
[make-child](#), 10
[make-code-transformer](#), 68
[make-outline](#), 54
[make-pict](#), 10
[make-pict-drawer](#), 47
[make-slide-inset](#), 60
 Making Pictures, 9
 Making Slides, 50
[margin](#), 57
 Merging Animations, 74
 Miscellaneous, 46
 More Pict Constructors, 33
[most-recent-slide](#), 55
 Mr. Potatohead, 41
[mzscheme-const-list](#), 67
[o-bullet](#), 57
[outline-flash](#), 45
[panorama](#), 31
[para](#), 52
[pict](#), 10
[pict](#), 9
 Pict Combiners, 22
 Pict Datatype, 9
 Pict Drawing Adjusters, 27
 Pict Finders, 31
[pict->bitmap](#), 47
[pict->code-pict](#), 69
[pict-ascent](#), 10
[pict-children](#), 10
[pict-convert](#), 49
[pict-convertible?](#), 48
[pict-descent](#), 10
[pict-draw](#), 10
[pict-height](#), 10
[pict-last](#), 10
[pict-panbox](#), 10
[pict-path?](#), 33
 Pict-Staging Helper, 60
[pict-width](#), 10
[pict?](#), 10
[pin-arrow-line](#), 19
[pin-arrows-line](#), 20
[pin-balloon](#), 39
[pin-line](#), 18
[pin-over](#), 26
[pin-under](#), 26
[pip-arrow-line](#), 18
[pip-arrows-line](#), 18
[pip-line](#), 17
[pip-wrap-balloon](#), 39
[play](#), 71
[play-n](#), 72
 Primary Slide Functions, 50
 Printing, 8
[printing?](#), 58
[prop:code-transformer](#), 68
[prop:pict-convertible](#), 48
[prop:pict-convertible?](#), 48
[racket/base-const-list](#), 66
[rb-find](#), 32
[rb-superimpose](#), 24
[rbl-find](#), 32
[rbl-superimpose](#), 24
[rc-find](#), 32
[rc-superimpose](#), 24
[re-slide](#), 55
[rectangle](#), 15
[refocus](#), 31
 Rendering, 47
[retract-most-recent-slide](#), 55
[reverse-animations](#), 75
[rotate](#), 28
[rounded-rectangle](#), 16
[rt](#), 51
[rt-find](#), 32
[rt-superimpose](#), 24
[rtl-find](#), 32
[rtl-superimpose](#), 24
[scale](#), 27
[scale-color](#), 46

[scale-to-fit](#), 27
[scale/improve-new-text](#), 29
[sequence-animations](#), 74
[set-margin!](#), 60
[set-page-numbers-visible!](#), 56
[set-title-h!](#), 60
[set-use-background-frame!](#), 55
[show-pict](#), 47
[size-in-pixels](#), 54
[slide](#), 50
Slide Basics, 4
Slide Registration, 54
[slide->pict](#), 55
[slide-inset?](#), 60
[slide-pict](#), 74
[slide?](#), 54
Slides to Picts, 61
slideshow, 1
slideshow/balloon, 38
slideshow/base, 50
slideshow/code, 63
slideshow/code-pict, 63
slideshow/face, 41
slideshow/flash, 44
slideshow/pict, 9
slideshow/pict-convert, 48
slideshow/play, 71
slideshow/slides-to-picts, 61
slideshow/start, 7
slideshow/step, 60
Slideshow: Figure and Presentation Tools, 1
[split-phase](#), 76
Staging Slides, 5
[standard-fish](#), 34
[start-at-recent-slide](#), 55
Stretching and Squashing Time, 75
[struct:child](#), 10
[struct:pict](#), 10
[subitem](#), 53
t, 51
table, 27
text, 12
[text-style/c](#), 21
[thermometer](#), 37
title-h, 58
titleless-page, 57
titlet, 52
tt, 51
[typeset-code](#), 63
Typesetting Racket Code, 63
[use-last](#), 31
[use-last*](#), 31
[vc-append](#), 22
Viewer Control, 55
[vl-append](#), 22
vline, 14
[vr-append](#), 22
with-steps, 60
with-steps~, 61
[wrap-balloon](#), 38