

Unstable GUI: May Change Without Warning

Version 5.93

January 29, 2014

This manual documents GUI libraries available in the [unstable](#) collection. See *Unstable: May Change Without Warning* for more information about unstable libraries.

1 Notify-boxes

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(require unstable/gui/notify)      package: gui-lib
```

```
notify-box% : class?  
  superclass: object%
```

A notify-box contains a mutable cell. The notify-box notifies its listeners when the contents of the cell is changed.

Examples:

```
> (define nb (new notify-box% (value 'apple)))  
  
> (send nb get)  
'apple  
> (send nb set 'orange)  
  
> (send nb listen (lambda (v) (printf "New value: ~s\n" v)))  
  
> (send nb set 'potato)  
New value: potato
```

```
(new notify-box% [value value]) → (is-a?/c notify-box%)  
  value : any/c
```

Creates a notify-box initially containing *value*.

```
(send a-notify-box get) → any/c
```

Gets the value currently stored in the notify-box.

```
(send a-notify-box set v) → void?  
  v : any/c
```

Updates the value stored in the notify-box and notifies the listeners.

```
(send a-notify-box listen listener) → void?  
  listener : (-> any/c any)
```

Adds a callback to be invoked on the new value when the notify-box's contents change.

```
(send a-notify-box remove-listener listener) → void?  
listener : (-> any/c any)
```

Removes a previously-added callback.

```
(send a-notify-box remove-all-listeners) → void?
```

Removes all previously registered callbacks.

```
(notify-box/pref proc  
  [#:readonly? readonly?]) → (is-a?/c notify-box%)  
proc : (case-> (-> any/c) (-> any/c void?))  
readonly? : boolean? = #f
```

Creates a notify-box with an initial value of (*proc*). Unless *readonly?* is true, *proc* is invoked on the new value when the notify-box is updated.

Useful for tying a notify-box to a preference or parameter. Of course, changes made directly to the underlying parameter or state are not reflected in the notify-box.

Examples:

```
> (define animal (make-parameter 'ant))  
  
> (define nb (notify-box/pref animal))  
  
> (send nb listen (lambda (v) (printf "New value: ~s\n" v)))  
  
> (send nb set 'bee)  
New value: bee  
  
> (animal 'cow)  
  
> (send nb get)  
'bee  
> (send nb set 'deer)  
New value: deer  
  
> (animal)  
'deer
```

```
(define-notify name value-expr)  
value-expr : (is-a?/c notify-box%)
```

Class-body form. Declares *name* as a field and *get-name*, *set-name*, and *listen-name* as methods that delegate to the *get*, *set*, and *listen* methods of *value*.

The *value-expr* argument must evaluate to a *notify-box*, not just the initial contents for a *notify box*.

Useful for aggregating many *notify-boxes* together into one “configuration” object.

Examples:

```
> (define config%
  (class object%
    (define-notify food (new notify-box% (value 'apple)))
    (define-notify animal (new notify-box% (value 'ant)))
    (super-new)))

> (define c (new config%))

> (send c listen-food
  (lambda (v) (when (eq? v 'honey) (send c set-
    animal 'bear))))

> (let ([food (get-field food c)])
  (send food set 'honey))

> (send c get-animal)
'bear
```

```
(menu-option/notify-box parent
  label
  notify-box)
→ (is-a?/c checkable-menu-item%)
parent : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
label : label-string?
notify-box : (is-a?/c notify-box%)
```

Creates a *checkable-menu-item%* tied to *notify-box*. The menu item is checked whenever *(send notify-box get)* is true. Clicking the menu item toggles the value of *notify-box* and invokes its listeners.

```
(checkbox/notify-box parent
  label
  notify-box) → (is-a?/c checkbox%)
parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
  (is-a?/c panel%) (is-a?/c pane%))
label : label-string?
notify-box : (is-a?/c notify-box%)
```

Creates a `check-box%` tied to `notify-box`. The check-box is checked whenever `(send notify-box get)` is true. Clicking the check box toggles the value of `notify-box` and invokes its listeners.

```
(choice/notify-box parent
      label
      choices
      notify-box) → (is-a?/c choice%)
parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)
          (is-a?/c panel%) (is-a?/c pane%))
label : label-string?
choices : (listof label-string?)
notify-box : (is-a?/c notify-box%)
```

Creates a `choice%` tied to `notify-box`. The choice control has the value `(send notify-box get)` selected, and selecting a different choice updates `notify-box` and invokes its listeners.

If the value of `notify-box` is not in `choices`, either initially or upon an update, an error is raised.

```
(menu-group/notify-box parent
      labels
      notify-box)
→ (listof (is-a?/c checkable-menu-item%))
parent : (or/c (is-a?/c menu%) (is-a?/c popup-menu%))
labels : (listof label-string?)
notify-box : (is-a?/c notify-box%)
```

Returns a list of `checkable-menu-item%` controls tied to `notify-box`. A menu item is checked when its label is `(send notify-box get)`. Clicking a menu item updates `notify-box` to its label and invokes `notify-box`'s listeners.

2 Preferences

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(require unstable/gui/prefs)      package: gui-lib
```

```
(pref:get/set pref) → (case-> (-> any/c) (-> any/c void?))  
  pref : symbol?
```

Returns a procedure that when applied to zero arguments retrieves the current value of the preference (`framework/preferences`) named `pref` and when applied to one argument updates the preference named `pref`.

3 Pict Utilities

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(require unstable/gui/pict)      package: unstable-lib
```

The functions and macros exported by this module are also exported by `unstable/gui/slideshow`.

3.1 Pict Colors

```
(color c p) → pict?  
  c : color/c  
  p : pict?
```

Applies color `c` to picture `p`. Equivalent to `(colorize p c)`.

Example:

```
> (color "red" (disk 20))
```



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

```
(magenta pict) → pict?  
  pict : pict?
```

These functions apply appropriate colors to picture *p*.

Example:

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



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

These functions produce lighter or darker versions of a color.

Example:

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



```
color/c : flat-contract?
```

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

3.2 Pict Manipulation

```
(fill pict width height) → pict?  
  pict : pict?  
  width : (or/c real? #f)  
  height : (or/c real? #f)
```

Extends *pict*'s bounding box to a minimum *width* and/or *height*, placing the original picture in the middle of the space.

Example:




```
> (frame (fill (disk 20) 40 40))
```




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

Scales *pict* so that its width and height are at most *width* and *height*, respectively. If *mode* is *'preserve*, the width and height are scaled by the same factor so *pict*'s aspect ratio is preserved; the result's bounding box may be smaller than *width* by *height*. If *mode* is *'inset*, the aspect ratio is preserved as with *'preserve*, but the resulting pict is centered in a bounding box of exactly *width* by *height*. If *mode* is *'distort*, the width and height are scaled separately.

Examples:

```
> (frame (scale-to (circle 100) 40 20))

> (frame (scale-to (circle 100) 40 20 #:mode 'inset))

> (frame (scale-to (circle 100) 40 20 #:mode 'distort))

```

3.2.1 Conditional Manipulations

These pict transformers all take boolean arguments that determine whether to transform the pict or leave it unchanged. These transformations can be useful for staged slides, as the resulting pict always has the same size and shape, and its contents always appear at the same position, but changing the boolean argument between slides can control when the transformation occurs.

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

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

```
(strike pict [strike?]) → pict?
  pict : pict?
  strike? : truth/c = #t
```

Displays a strikethrough image by putting a line through the middle of *pict* if *strike?* is true; produces *pict* unchanged otherwise.

Example:

```
> (strike (colorize (disk 20) "yellow"))
```



```
(shade pict [shade? #:ratio ratio]) → pict?
  pict : pict?
  shade? : truth/c = #t
  ratio : (real-in 0 1) = 1/2
```

Shades *pict* to show with *ratio* of its normal opacity; if *ratio* is 1 or *shade?* is #f, shows *pict* unchanged.

Example:

```
> (shade (colorize (disk 20) "red"))
```



3.2.2 Conditional Combinations

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

```
(pict-if maybe-combine test-expr then-expr else-expr)
  maybe-combine =
    | #:combine combine-expr
```

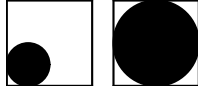
Chooses either *then-expr* or *else-expr* based on *test-expr*, similarly to *if*. Combines the chosen, visible image with the other, invisible image using *combine-expr*, defaulting to *pict-combine*.

Example:

```

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

```



```

(pict-cond maybe-combine [test-expr pict-expr] ...)
maybe-combine =
  | #:combine combine-expr

```

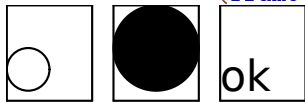
Chooses a *pict-expr* based on the first successful *test-expr*, similarly to *cond*. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to *pict-combine*.

Example:

```

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

```



```

(pict-case test-expr maybe-combine [literals pict-expr] ...)
maybe-combine =
  | #:combine combine-expr

```

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

Example:

```

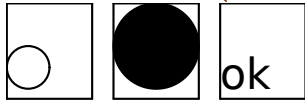
> (let ([f (lambda (x)

```

```

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

```



```

(pict-match test-expr maybe-combine [pattern pict-expr] ...)
maybe-combine =
  | #:combine combine-expr

```

Chooses a *pict-expr* based on *test-expr* and each *pattern*, similarly to *match*. Combines the chosen, visible image with the other, invisible images using *combine-expr*, defaulting to *pict-combine*.

pict-combine

This syntax parameter determines the default pict combining form used by the above macros. It defaults to *lbl-superimpose*.

(with-pict-combine *combine-id* *body* ...)

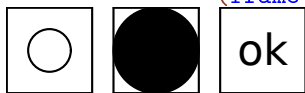
Sets *pict-combine* to refer to *combine-id* within each of the *body* terms, which are spliced into the containing context.

Example:

```

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

```



3.3 Shapes with Borders

The subsequent bindings were added by Vincent St-Amour.

```
(ellipse/border w
                h
                [#:color color
                 #:border-color border-color
                 #:border-width border-width]) → pict?

w : real?
h : real?
color : color/c = "white"
border-color : color/c = "black"
border-width : real? = 2
(circle/border diameter
             [#:color color
              #:border-color border-color
              #:border-width border-width]) → pict?

diameter : real?
color : color/c = "white"
border-color : color/c = "black"
border-width : real? = 2
(rectangle/border w
                  h
                  [#:color color
                   #:border-color border-color
                   #:border-width border-width]) → pict?

w : real?
h : real?
color : color/c = "white"
border-color : color/c = "black"
border-width : real? = 2
(rounded-rectangle/border w
                          h
                          [#:color color
                           #:border-color border-color
                           #:border-width border-width
                           #:corner-radius corner-radius
                           #:angle angle])

→ pict?
w : real?
h : real?
color : color/c = "white"
border-color : color/c = "black"
border-width : real? = 2
corner-radius : real? = -0.25
angle : real? = 0
```

These functions create shapes with border of the given color and width.

Examples:

```
> (ellipse/border 40 20 #:border-color "blue")
```



```
> (rounded-rectangle/border 40 20 #:color "red")
```



3.4 Lines with Labels

```
(pin-label-line label
                pict
                src-pict
                src-coord-fn
                dest-pict
                dest-coord-fn
                [#: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?
                #:x-adjust x-adjust
                #:y-adjust y-adjust]) → pict?

label : pict?
pict : pict?
src-pict : pict-path?
src-coord-fn : (-> pict-path? (values real? real?))
dest-pict : pict-path?
dest-coord-fn : (-> 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 real? #f) = #f
color : (or/c #f string? (is-a?/c color%)) = #f
alpha : (real-in 0 1) = 1
style : pen-style/c = 'solid
under? : any/c = #f
x-adjust : real? = 0
y-adjust : real? = 0
```

The subsequent bindings were added by Scott Owens.

```

(pin-arrow-label-line label
  arrow-size
  pict
  src-pict
  src-coord-fn
  dest-pict
  dest-coord-fn
  [#: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
   #:solid? solid?
   #:under? under?
   #:hide-arrowhead? hide-arrowhead?
   #:x-adjust x-adjust
   #:y-adjust y-adjust])
→ pict?
label : pict?
arrow-size : real?
pict : pict?
src-pict : pict-path?
src-coord-fn : (-> pict-path? (values real? real?))
dest-pict : pict-path?
dest-coord-fn : (-> 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 real? #f) = #f
color : (or/c #f string? (is-a?/c color%)) = #f
alpha : (real-in 0 1) = 1
style : pen-style/c = 'solid
solid? : boolean? = #t
under? : any/c = #f
hide-arrowhead? : any/c = #f
x-adjust : real? = 0
y-adjust : real? = 0

```

```

(pin-arrows-label-line label
  arrow-size
  pict
  src-pict
  src-coord-fn
  dest-pict
  dest-coord-fn
  [#: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
   #:solid? solid?
   #:under? under?
   #:hide-arrowhead? hide-arrowhead?
   #:x-adjust x-adjust
   #:y-adjust y-adjust])
→ pict?
label : pict?
arrow-size : real?
pict : pict?
src-pict : pict-path?
src-coord-fn : (-> pict-path? (values real? real?))
dest-pict : pict-path?
dest-coord-fn : (-> 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 real? #f) = #f
color : (or/c #f string? (is-a?/c color%)) = #f
alpha : (real-in 0 1) = 1
style : pen-style/c = 'solid
solid? : boolean? = #t
under? : any/c = #f
hide-arrowhead? : any/c = #f
x-adjust : real? = 0
y-adjust : real? = 0

```

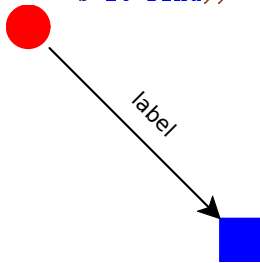
These functions behave like `pin-line`, `pin-arrow-line` and `pin-arrows-line` with the addition of a label attached to the line.

Example:


```

> (let* ([a (red (disk 20))]
         [b (blue (filled-rectangle 20 20))]
         [p (vl-append a (hb-append (blank 100) b))])
  (pin-arrow-label-line
   (rotate (text "label" null 10) (/ pi -4))
   10 p
   a rb-find
   b lt-find))

```



3.5 Blur

```

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

```

The subsequent bindings were added by Ryan Culpepper.

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

The `blur` function takes work proportional to

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

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

Examples:

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

blur

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

more blur

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

much blur

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

horiz. blur

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

Example:

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

more blur

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

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

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

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

Examples:

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

shadow

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

shadow

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

shadow

```
(blur-bitmap! bitmap h-radius [v-radius]) → void?
  bitmap : (is-a?/c bitmap%)
  h-radius : (and/c real? (not/c negative?))
  v-radius : (and/c real? (not/c negative?)) = h-radius
```

Blurs *bitmap* using blur radii *h-radius* and *v-radius*.

3.5.1 Tagged Picts

```
(tag-pict p tag) → pict?
  p : pict?
  tag : symbol?
```

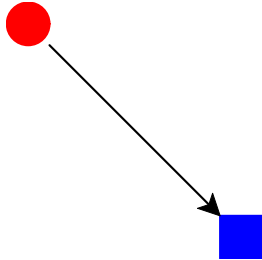
Returns a pict like *p* that carries a symbolic tag. The tag can be used with `find-tag` to locate the pict.

```
(find-tag p find) → (or/c pict-path? #f)
  p : pict?
  find : tag-path?
```

Locates a sub-pict of *p*. Returns a pict-path that can be used with functions like `lt-find`, etc.

Example:

```
> (let* ([a (tag-pict (red (disk 20)) 'a)]
         [b (tag-pict (blue (filled-rectangle 20 20)) 'b)]
         [p (vl-append a (hb-append (blank 100) b))])
      (pin-arrow-line 10 p
                     (find-tag p 'a) rb-find
                     (find-tag p 'b) lt-find))
```

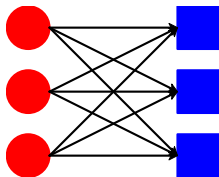


```
(find-tag* p find) → (listof pict-path?)
p : pict?
find : tag-path?
```

Like `find-tag`, but returns all pict-paths corresponding to the given tag-path.

Example:

```
> (let* ([a (lambda () (tag-pict (red (disk 20)) 'a))]
         [b (lambda () (tag-pict (blue (filled-
rectangle 20 20)) 'b))]
         [as (vc-append 10 (a) (a) (a))]
         [bs (vc-append 10 (b) (b) (b))]
         [p (hc-append as (blank 60 0) bs)])
      (for*/fold ([p p])
                 ([apath (in-list (find-tag* p 'a))]
                  [bpath (in-list (find-tag* p 'b))])
                 (pin-arrow-line 4 p
                                apath rc-find
                                bpath lc-find)))
```



```
(tag-path? x) → boolean?
x : any/c
```

Returns `#t` if `x` is a symbol or a non-empty list of symbols, `#f` otherwise.

3.6 Shadow Frames

```
(shadow-frame pict
  ...
  [#:sep separation
   #:margin margin
   #:background-color bg-color
   #:frame-color frame-color
   #:frame-line-width frame-line-width
   #:shadow-side-length shadow-side-length
   #:shadow-top-y-offset shadow-top-y-offset
   #:shadow-bottom-y-offset shadow-bottom-y-offset
   #:shadow-descent shadow-descent
   #:shadow-alpha-factor shadow-alpha-factor
   #:blur blur-radius])
→ pict?
pict : pict?
separation : real? = 5
margin : real? = 20
bg-color : (or/c string? (is-a?/c color%)) = "white"
frame-color : (or/c string? (is-a?/c color%)) = "gray"
frame-line-width : (or/c real? #f) = 0
shadow-side-length : real? = 4
shadow-top-y-offset : real? = 10
shadow-bottom-y-offset : real? = 4
shadow-descent : (and/c real? (not/c negative?)) = 40
shadow-alpha-factor : real? = 3/4
blur-radius : (and/c real? (not/c negative?)) = 20
```

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

The *picts* are vertically appended with *separation* space between them. They are placed on a rectangular background of solid *bg-color* with *margin* space on all sides. A frame of *frame-color* and *frame-line-width* is added around the rectangle. The rectangle casts a shadow that extends *shadow-side-length* to the left and right, starts *shadow-top-y-offset* below the top of the rectangle and extends to *shadow-bottom-y-offset* below the bottom of the rectangle in the center and an additional *shadow-descent* below that on the sides. The shadow is painted using a linear gradient; *shadow-alpha-factor* determines its density at the center. Finally, the shadow is blurred by *blur-radius*; all previous measurements are pre-blur measurements.

Example:

```
> (scale (shadow-frame (text "text in a nifty
frame" null 60)) 1/2)
```

text in a nifty frame

```
(arch outer-width
      inner-width
      solid-height
      leg-height) → pict?
outer-width : real?
inner-width : real?
solid-height : real?
leg-height : real?
```

Creates an arch.

Example:

```
> (colorize (arch 100 80 20 20) "red")
```



3.7 Additional combinators

```
(backdrop pict [#:color color]) → pict?
pict : pict?
color : color/c = "white"
```

Adds a background highlighted with *color* to *pict*.

Examples:

```
> (backdrop (circle 20) #:color "whitesmoke")
```



```
> (backdrop (text "broccoli rabé") #:color "PaleGreen")
broccoli rabé
```

```
(cross-out pict
  [#:width width
   #:style style
   #:color color]) → pict?
```

The subsequent bindings were added by Asumu Takikawa.

```

pict : pict?
width : real? = 1
      (or/c 'transparent 'solid 'xor
style :      'hilite 'dot 'long-dash 'short-dash = 'solid
          'dot-dash 'xor-dot 'xor-long-dash
          'xor-short-dash 'xor-dot-dash)
color : color/c = "black"

```

Crosses out *pict* with two diagonal lines drawn with the given line *width* and with the line *style*. The lines are colored with *color*.

Examples:

```
> (cross-out (circle 20))
```



```
> (cross-out (rectangle 30 20) #:width 2 #:style 'long-dash)
```



```
> (cross-out (text "rapini") #:width 3 #:color "red")
```



4 Slideshow Presentations

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(require unstable/gui/slideshow)      package: unstable-lib
```

This module also exports everything provided by `unstable/gui/pict`.

4.1 Text Formatting

```
(with-size size expr)
```

Sets `current-font-size` to *size* while running *expr*.

```
(with-scale scale expr)
```

Multiplies `current-font-size` by *scale* while running *expr*.

```
(big text)  
(small text)
```

Scale `current-font-size` by $3/2$ or $2/3$, respectively, while running *text*.

```
(with-font font expr)
```

Sets `current-main-font` to *font* while running *expr*.

```
(with-style style expr)
```

Adds *style* to `current-main-font` (via `cons`) while running *expr*.

```
(bold text)  
(italic text)  
(subscript text)  
(superscript text)  
(caps text)
```

Adds the attributes for bold, italic, superscript, subscript, or small caps text, respectively, to `current-main-font` while running *text*.

4.2 Tables

```
(tabular row
  ...
  [:gap gap
   #:hgap hgap
   #:vgap vgap
   #:align align
   #:halign halign
   #:valign valign]) → pict?
row : (listof (or/c string? pict?))
gap : natural-number/c = gap-size
hgap : natural-number/c = gap
vgap : natural-number/c = gap
align : (->* [] [] #:rest (listof pict?) pict?)
        = lbl-superimpose
halign : (->* [] [] #:rest (listof pict?) pict?) = align
valign : (->* [] [] #:rest (listof pict?) pict?) = align
```

Constructs a table containing the given *rows*, all of which must be of the same length. Applies **t** to each string in a *row* to construct a pict. The *hgap*, *vgap*, *halign*, and *valign* are used to determine the horizontal and vertical gaps and alignments as in **table** (except that every row and column is uniform).

4.3 Multiple Columns

```
(two-columns one two)
```

Constructs a two-column pict using *one* and *two* as the two columns. Sets **current-para-width** appropriately in each column.

```
(mini-slide pict ...) → pict?
pict : pict?
```

Appends each *pict* vertically with space between them, similarly to the **slide** function.

```
(columns pict ...) → pict?
pict : pict?
```

Combines each *pict* horizontally, aligned at the top, with space in between.

```
(column width body ...)
```

Sets `current-para-width` to `width` during execution of the `body` expressions.

```
(column-size n [r]) → real?  
  n : exact-positive-integer?  
  r : real? = (/ n)
```

Computes the width of one column out of `n` that takes up a ratio of `r` of the available space (according to `current-para-width`).

4.4 Staged Slides

```
(staged [name ...] body ...)
```

Executes the `body` terms once for each stage `name`. The terms may include expressions and mutually recursive definitions. Within the body, each `name` is bound to a number from 1 to the number of stages in order. Furthermore, during execution `stage` is bound to the number of the current stage and `stage-name` is bound to a symbol representing the `name` of the current stage. By comparing `stage` to the numeric value of each `name`, or `stage-name` to quoted symbols of the form `'name`, the user may compute based on the progression of the stages.

```
stage  
stage-name
```

These keywords are bound during the execution of `staged` and should not be used otherwise.

```
(slide/staged [name ...] arg ...)
```

Creates a staged slide. Equivalent to `(staged [name ...] (slide arg ...))`.

Within a staged slide, the boolean arguments to `hide`, `show`, `strike`, and `shade` can be used to determine in which stages to perform a transformation. The macros `pict-if`, `pict-cond`, `pict-case`, and `pict-match` may also be used to create images which change naturally between stages.

4.5 Revealing Slides

```
(reveal number expr ...)
```

Expands to either `(show expr ...)` or `(hide expr ...)` if `number` is greater than or equal to the current revealed slide within a revealing-slide.

The subsequent bindings were added by Jon Rafkind.

```
| (revealing-slide expr ...)
```

Creates N slides where N is the maximum number given to a reveal expression as the first argument. Each slide has the current reveal number incremented by one so progressive slides can reveal parts in that appear in arbitrary places.

```
(revealing-slide
  (hc-append (reveal 0 (t "I show up first"))
             (reveal 1 (t "I show up second"))))
(reveal 1 (t "I also show up second"))
```

```
| (items-slide (name ...) expr ...)
```

Creates N slides where N is the maximum number given to a reveal similar to revealing-slide. Each slide will show the names on the left hand side using right justification and only one reveal will be displayed on the right. The order of the names will be matched with the current reveal number so the first item will be displayed while the first reveal is displayed. Reveals that are not active will have their corresponding items displayed but in a light font so as to indicate inactivity.

```
(items-slide ("item1" "item2" "item3")
  (reveal 0
    (t "I will show up for item1"))
  (reveal 1
    (t "I will show up for item2"))
  (reveal 2
    (t "I will show up for item3")))
```

4.6 Miscellaneous Slide Utilities

```
| (blank-line) → pict?
```

Adds a blank line of the current font size's height.

The subsequent bindings were added by Scott Owens.

5 Progressive Picts and Slides

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

5.1 Progressive Picts

```
(require unstable/gui/ppict)      package: unstable-lib
```

A *progressive pict* or “ppict” is a kind of `pict` that has an associated “pict placer,” which generally represents a position and alignment. New picts can be placed on the progressive pict by calling `ppict-add`, and the placer can be updated by calling `ppict-go`. The `ppict-do` form provides a compact notation for sequences of those two operations.

```
(ppict-do base-expr ppict-do-fragment ...)  
(ppict-do* base-expr ppic-do-fragment ...)  
  
ppict-do-fragment = #:go placer-expr  
                  | #:set pict-expr  
                  | #:next  
                  | #:alt (ppict-do-fragment ...)  
                  | elem-expr  
  
base-expr : pict?  
placer-expr : placer?  
pict-expr : pict?  
elem-expr : (or/c pict? real? #f)
```

Builds a pict (and optionally a list of intermediate picts) progressively. The `ppict-do` form returns only the final pict; any uses of `#:next` are ignored. The `ppict-do*` form returns two values: the final pict and a list of all partial picts emitted due to `#:next` (the final pict is not included).

A `#:go` fragment changes the current placer. A `#:set` fragment replaces the current pict state altogether with a new computed pict. A `#:next` fragment saves a pict including only the contents emitted so far (but whose alignment takes into account picts yet to come). A `#:alt` fragment saves the current pict state, executes the sub-sequence that follows, saves the result (as if the sub-sequence ended with `#:next`), then restores the saved pict state before continuing.

The `elem-exprs` are interpreted by the current placer. A numeric `elem-expr` usually represents a spacing change, but some placers do not support them. A spacing change only

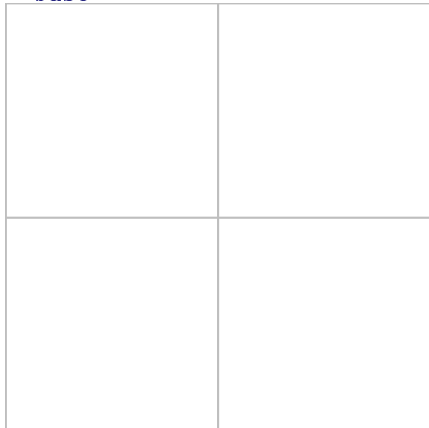
affects added pict's up until the next placer is installed; when a new placer is installed, the spacing is reset, usually to 0.

The `ppict-do-state` form tracks the current state of the pict. It is updated before a `#:go` or `#:set` fragment or before a sequence of `elem-exprs`. It is not updated in the middle of a chain of `elem-exprs`, however.

Examples:

```
> (define base
  (ppict-do (colorize (rectangle 200 200) "gray")
            #:go (coord 1/2 1/2 'cc)
            (colorize (hline 200 1) "gray")
            #:go (coord 1/2 1/2 'cc)
            (colorize (vline 1 200) "gray")))
```

```
> base
```



The use of `ppict-do` in the definition of `base` above is equivalent to

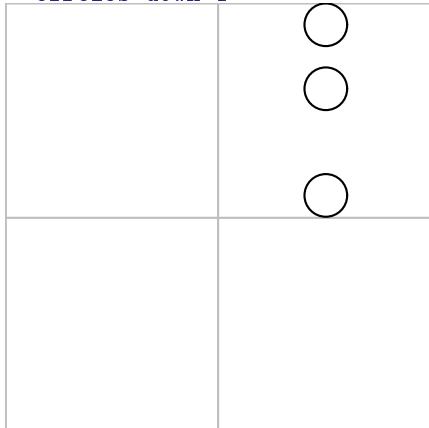
```
(let* ([pp (colorize (rectangle 200 200) "gray")]
       [pp (ppict-go pp (coord 1/2 1/2 'cc))]
       [pp (ppict-add pp (colorize (hline 200 1) "gray"))]
       [pp (ppict-go pp (coord 1/2 1/2 'cc))]
       [pp (ppict-add pp (colorize (vline 1 200) "gray"))])
  pp)
```

Examples:

```
> (define circles-down-1
  (ppict-do base
            #:go (grid 2 2 2 1 'ct)
            10))
```

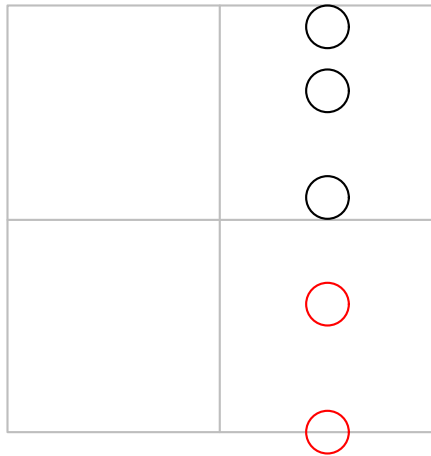
```
(circle 20)
(circle 20)
30
(circle 20)))
```

```
> circles-down-1
```

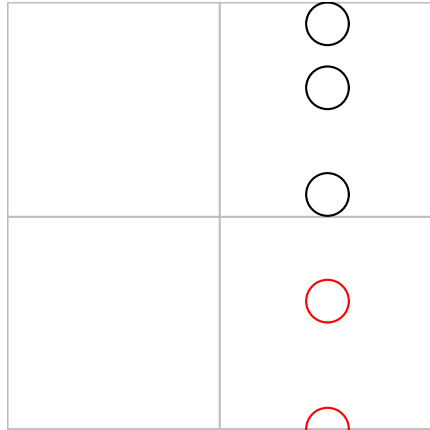


```
> (define circles-down-2
  (ppict-do circles-down-1
    (colorize (circle 20) "red")
    40
    (colorize (circle 20) "red")))
```

```
> (inset circles-down-2 20) ; draws outside its bounding box
```



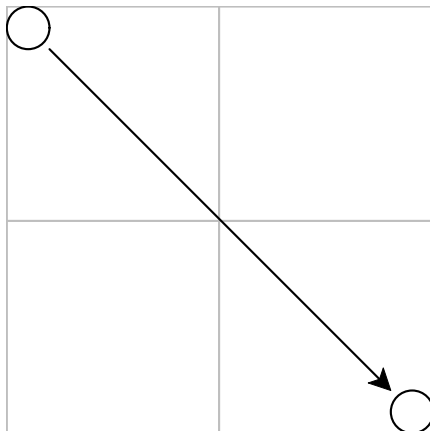
```
> (inset (clip circles-down-2) 20)
```



```

> (ppict-do base
   #:go (coord 0 0 'lt)
   (tag-pict (circle 20) 'circA)
   #:go (coord 1 1 'rb)
   (tag-pict (circle 20) 'circB)
   #:set (let ([p ppict-do-state])
           (pin-arrow-line 10 p
                           (find-tag p 'circA) rb-find
                           (find-tag p 'circB) lt-find)))

```



```

> (let-values ([ (final intermediates)
                 (ppict-do* base
                           #:go (coord 1/4 1/2 'cb)
                           (text "shapes:")
                           #:go (coord 1/2 1/2 'lb)
                           #:alt [(circle 20)]
                           #:alt [(rectangle 20 20)]

```

```

                                (text "and more!"))]]
(append intermediates (list final)))

```

shapes:	shapes:	shapes: and more!
---------	---------	-------------------

```

, (

```

More examples of `ppict-do` are scattered throughout this section.

ppict-do-state

Tracks the current state of a `ppict-do` or `ppict-do*` form.

```

(ppict? x) → boolean?
  x : any/c

```

Returns `#t` if `x` is a progressive pict, `#f` otherwise.

```

(ppict-go p pl) → ppict?
  p : pict?
  pl : placer?

```

Creates a progressive pict with the given base pict `p` and the placer `pl`.

```

(ppict-add pp elem ...) → pict?
  pp : ppict?
  elem : (or/c pict? real? #f 'next)
(ppict-add* pp elem ...) → pict? (listof pict?)
  pp : ppict?
  elem : (or/c pict? real? #f 'next)

```

Creates a new pict by adding each `elem` pict on top of `pp` according to `pp`'s placer. The result pict may or may not be a progressive pict, depending on the placer used. The `ppict-add` function only the final pict; any occurrences of `'next` are ignored. The `ppict-add*` function returns two values: the final pict and a list of all partial picts emitted due to `'next` (the final pict is not included).

An *elem* that is a real number changes the spacing for subsequent additions. A *elem* that is *#f* is discarded; it is permitted as a convenience for conditionally including sub-picts. Note that *#f* is not equivalent to `(blank 0)`, since the latter will cause spacing to be added around it.

```
(placer? x) → boolean?
  x : any/c
```

Returns *#t* if *x* is a placer, *#f* otherwise.

```
(refpoint-placer? x) → boolean?
  x : any/c
```

Returns *#t* if *x* is a placer based on a reference point, *#f* otherwise.

```
(coord rel-x
      rel-y
      [align
       #:abs-x abs-x
       #:abs-y abs-y
       #:compose composer]) → refpoint-placer?
rel-x : real?
rel-y : real?
align : (or/c 'lt 'ct 'rt 'lc 'cc 'rc 'lb 'cb 'rb) = 'cc
abs-x : real? = 0
abs-y : real? = 0
composer : procedure? = computed from align
```

Returns a placer that places pict according to *rel-x* and *rel-y*, which are interpreted as fractions of the width and height of the base progressive pict. That is, `0, 0` is the top left corner of the base's bounding box, and `1, 1` is the bottom right. Then *abs-x* and *abs-y* offsets are added to get the final reference point.

Additions are aligned according to *align*, a symbol whose name consists of a horizontal alignment character followed by a vertical alignment character. For example, if *align* is `'lt`, the pict is placed so that its left-top corner is at the reference point; if *align* is `'rc`, the pict is placed so that the center of its bounding box's right edge coincides with the reference point.

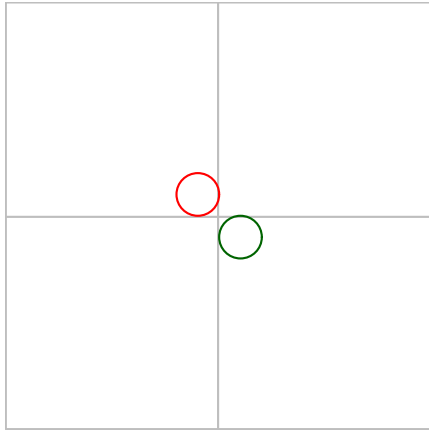
By default, if there are multiple picts to be placed, they are vertically appended, aligned according to the horizontal component of *align*. For example, if *align* is `'cc`, the default *composer* is `vc-append`; for `'lt`, the default *composer* is `vl-append`. The spacing is initially `0`.

Examples:

```

> (ppict-do base
  #:go (coord 1/2 1/2 'rb)
  (colorize (circle 20) "red")
  #:go (coord 1/2 1/2 'lt)
  (colorize (circle 20) "darkgreen"))

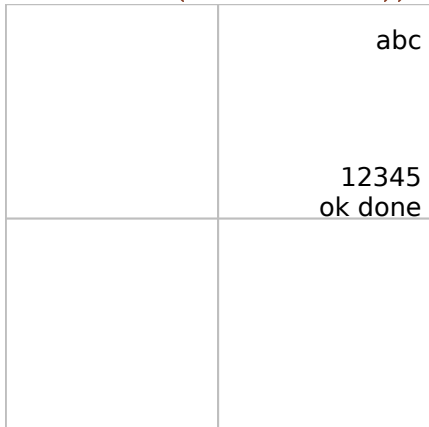
```



```

> (ppict-do base
  #:go (coord 1 0 'rt #:abs-x -5 #:abs-y 10)
  50 ; change spacing
  (text "abc")
  (text "12345")
  0 ; and again
  (text "ok done"))

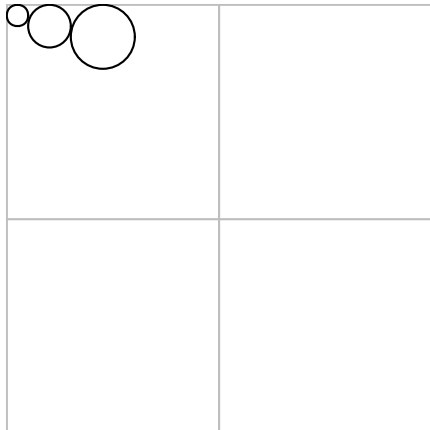
```



```

> (ppict-do base
  #:go (coord 0 0 'lt #:compose ht-append)
  (circle 10)
  (circle 20)
  (circle 30))

```



```
(grid cols
      rows
      col
      row
      [align
       #:abs-x abs-x
       #:abs-y abs-y
       #:compose composer]) → refpoint-placer?
cols : exact-positive-integer?
rows : exact-positive-integer?
col  : exact-integer?
row  : exact-integer?
align : (or/c 'lt 'ct 'rt 'lc 'cc 'rc 'lb 'cb 'rb) = 'cc
abs-x : real? = 0
abs-y : real? = 0
composer : procedure? = computed from align
```

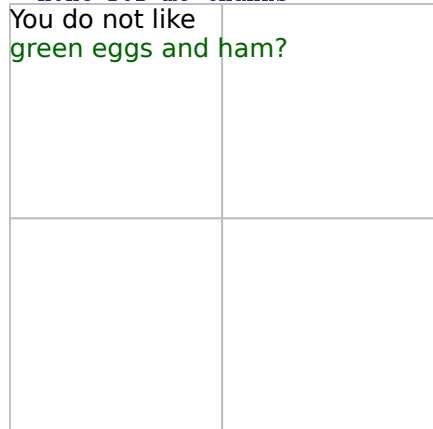
Returns a placer that places pict's according to a position in a virtual grid. The *row* and *col* indexes are numbered starting at 1.

Uses of `grid` can be translated into uses of `coord`, but the translation depends on the alignment. For example, `(grid 2 2 1 1 'lt)` is equivalent to `(coord 0 0 'lt)`, but `(grid 2 2 1 1 'rt)` is equivalent to `(coord 1/2 0 'rt)`.

Examples:

```
> (define none-for-me-thanks
   (ppict-do base
     #:go (grid 2 2 1 1 'lt)
     (text "You do not like")
     (colorize (text "green eggs and
ham?" "darkgreen"))))
```

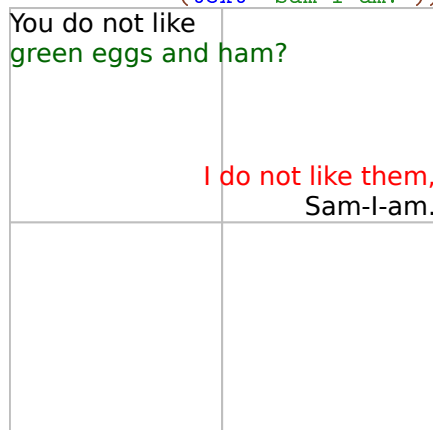
```
> none-for-me-thanks
You do not like
green eggs and ham?
```



```
> (ppict-do none-for-me-thanks
     #:go (grid 2 2 2 1 'rb)
     (colorize (text "I do not like them,") "red")
     (text "Sam-I-am."))
```

```
You do not like
green eggs and ham?

I do not like them,
Sam-I-am.
```



```
(cascade [step-x step-y]) → placer?
  step-x : (or/c real? 'auto) = 'auto
  step-y : (or/c real? 'auto) = 'auto
```

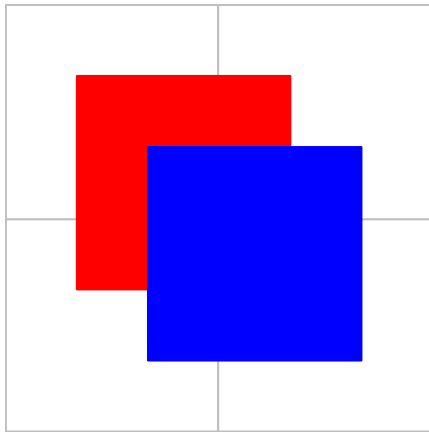
Returns a placer that places pict's by evenly spreading them diagonally across the base pict in “cascade” style. This placer does not support changing the spacing by including a real number within the pict sequence.

When a list pict's is to be placed, their bounding boxes are normalized to the maximum width and height of all pict's in the list; each pict is centered in its new bounding box. The pict's are then cascaded so there is *step-x* space between each of the pict's' left edges; there is also *step-x* space between the base pict's left edge and the first pict's left edge. Similarly for *step-y* and the vertical spacing.

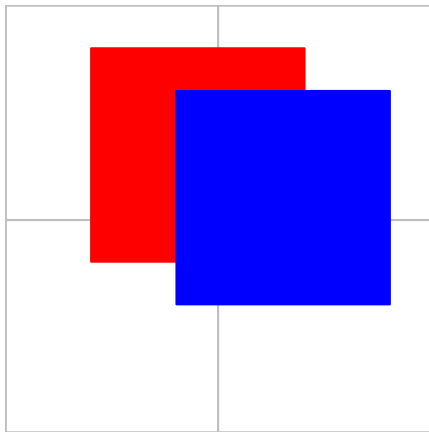
If `step-x` or `step-y` is `'auto`, the spacing between the centers of the pict to be placed is determined automatically so that the inter-pict spacing is the same as the spacing between the last pict and the base.

Examples:

```
> (ppict-do base
   #:go (cascade)
   (colorize (filled-rectangle 100 100) "red")
   (colorize (filled-rectangle 100 100) "blue"))
```



```
> (ppict-do base
   #:go (cascade 40 20)
   (colorize (filled-rectangle 100 100) "red")
   (colorize (filled-rectangle 100 100) "blue"))
```



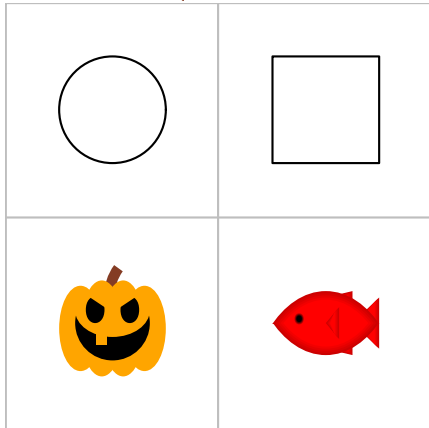
```
(tile cols rows) → placer?
  cols : exact-positive-integer?
  rows : exact-positive-integer?
```

Returns a placer that places pict by tiling them in a grid `cols` columns wide and `rows` rows

high.

Example:

```
> (ppict-do base
    #:go (tile 2 2)
    (circle 50)
    (rectangle 50 50)
    (jack-o-lantern 50)
    (standard-fish 50 30 #:color "red"))
```



```
(at-find-pict find-path
  [finder
   align
   #:abs-x abs-x
   #:abs-y abs-y
   #:compose composer]) → refpoint-placer?
find-path : (or/c tag-path? pict-path?)
finder : procedure? = cc-find
align : (or/c 'lt 'ct 'rt 'lc 'cc 'rc 'lb 'cb 'rb) = 'cc
abs-x : real? = 0
abs-y : real? = 0
composer : procedure? = computed from align
```

Returns a placer that places pict's according to a reference point based on an existing pict within the base.

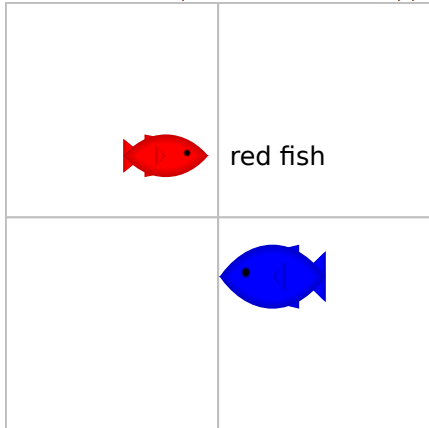
Example:

```
> (ppict-do base
    #:go (cascade)
    (tag-pict (standard-fish 40 20 #:direction 'right #:color "red") 'red-
    fish))
```

```

        (tag-pict (standard-fish 50 30 #:direction 'left #:color "blue") 'blue-
fish)
        #:go (at-find-pict 'red-fish rc-find 'lc #:abs-x 10)
        (text "red fish"))

```



```

(merge-refpoints x-placer y-placer) → refpoint-placer?
  x-placer : refpoint-placer?
  y-placer : refpoint-placer?

```

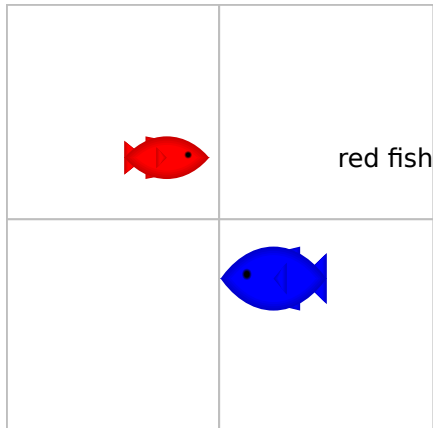
Returns a placer like *x-placer* except that the y-coordinate of its reference point is computed by *y-placer*.

Example:

```

> (ppict-do base
  #:go (cascade)
  (tag-pict (standard-fish 40 20 #:direction 'right #:color "red") 'red-
fish)
  (tag-pict (standard-fish 50 30 #:direction 'left #:color "blue") 'blue-
fish)
  #:go (merge-refpoints (coord 1 0 'rc)
    (at-find-pict 'red-fish))
  (text "red fish"))

```



5.2 Progressive Slides

```
(require unstable/gui/pslide) package: unstable-lib
```

```
(pslide ppict-do-fragment ...)
```

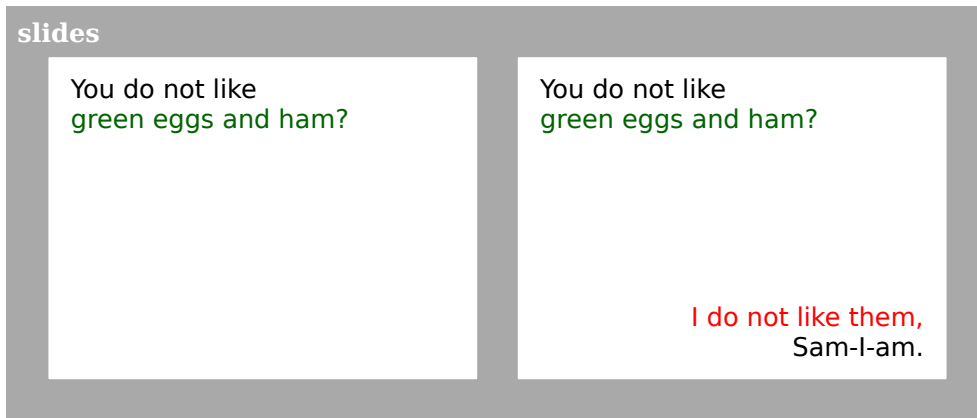
Produce slide(s) using progressive pict. See `ppict-do` for an explanation of *ppict-do-fragments*.

Note that like `slide` but unlike `ppict-do*`, the number of slides produced is one greater than the number of `#:next` uses; that is, a slide is created for the final pict.

Remember to include `gap-size` after updating the current placer if you want `slide`-like spacing.

Example:

```
> (pslide #:go (coord 0 0 'lt)
      (t "You do not like")
      (colorize (t "green eggs and ham?") "darkgreen")
      #:next
      #:go (coord 1 1 'rb)
      (colorize (t "I do not like them,") "red")
      (t "Sam-I-am."))
```

Note that the text is not flush against the sides of the slide, because `pslide` uses a base pict the size of the client area, excluding the margins.

```
(pslide-base-pict) → (-> pict)
(pslide-base-pict make-base-pict) → void?
  make-base-pict : (-> pict)
```

Controls the initial pict used by `pslide`. The default value is

```
(lambda () (blank client-w client-h))
```

```
(pslide-default-placer) → placer?
(pslide-default-placer placer) → void?
  placer : placer?
```

Controls the initial placer used by `pslide`. The default value is

```
(coord 1/2 1/2 'cc)
```

6 Snip Utilities

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(require unstable/gui/snip)      package: unstable-lib
```

```
snip-canvas% : class?  
  superclass: editor-canvas%
```

A canvas that contains a single snip.

Snips cannot be placed directly on dialogs, frames and panels. To use an interactive snip in a GUI, it must be inserted into an editor, which itself must be placed on a special canvas, which can be placed in a GUI container. To provide a seamless user experience, the editor should be enabled but not writable, not be able to receive focus, not have scrollbars, and other small details.

The `snip-canvas%` class handles these details, making it easy to use interactive snips as normal GUI elements.

```
(new snip-canvas%  
  [parent parent]  
  [make-snip make-snip]  
  [[style style]  
  [label label]  
  [horizontal-inset horizontal-inset]  
  [vertical-inset vertical-inset]  
  [enabled enabled]  
  [vert-margin vert-margin]  
  [horiz-margin horiz-margin]  
  [min-width min-width]  
  [min-height min-height]  
  [stretchable-width stretchable-width]  
  [stretchable-height stretchable-height]])  
→ (is-a?/c snip-canvas%)  
parent : (or/c (is-a?/c frame%) (is-a?/c dialog%)  
              (is-a?/c panel%) (is-a?/c pane%))  
make-snip : ((integer-in 0 10000) (integer-in 0 10000) . -> . snip%)  
           (listof (one-of/c 'no-border 'control-border 'combo  
                           'resize-corner 'no-focus 'deleted  
                           'transparent))  
           = null  
label : (or/c label-string? false/c) = #f  
horizontal-inset : (integer-in 0 1000) = 5
```

```

vertical-inset : (integer-in 0 1000) = 5
enabled : any/c = #t
vert-margin : (integer-in 0 1000) = 0
horiz-margin : (integer-in 0 1000) = 0
min-width : (integer-in 0 10000) = 0
min-height : (integer-in 0 10000) = 0
stretchable-width : any/c = #t
stretchable-height : any/c = #t

```

Unlike instances of `editor-canvas%`, each instance of this class creates and manages its own editor. The editor contains one object: a `snip%` instance created by `make-snip`.

The `make-snip` function receives the requested width and height of the snip, which are calculated from the size of the snip canvas. It is called the first time the snip canvas is resized, which most likely coincides with the first time the snip canvas is shown. The snip is thus created *lazily*: only when needed, at the size needed. See `on-size` for more details and an example.

The `style` list is prepended with `'no-hscroll` and `'no-vscroll` before being passed to the `editor-canvas%` constructor. The other constructor arguments are passed untouched.

```
(send a-snip-canvas get-snip) → (or/c (is-a?/c snip%) #f)
```

Returns the wrapped snip, or `#f` if `make-snip` has not been called yet.

```
(send a-snip-canvas on-size width height) → void?
width : (integer-in 0 10000)
height : (integer-in 0 10000)
```

Overrides `on-size` in `editor-canvas%`.

This is called when the snip canvas is resized.

On the first call, `on-size` calls `make-snip` with width and height arguments respectively `(max 0 (- width (* 2 horizontal-inset)))` and `(max 0 (- height (* 2 vertical-inset)))`. It then inserts the resulting snip into its editor.

On subsequent calls, `on-size` calls the snip's `resize` method, calculating the width and height arguments the same way.

When a `snip-canvas%` instance is intended to wrap an existing `snip%` instance, `make-snip` should simply resize it and return it.

Example: functions from `plot` create snips and call a function similar to the following to place plots in a frame:

```
(define (make-snip-frame snip w h label)
  (define (make-snip width height)
```

```
(send snip resize width height)
snip)

(define frame
  (new frame%
    [label label]
    [width (+ 5 5 5 5 w)]
    [height (+ 5 5 5 5 h)]))

(new snip-canvas%
  [parent frame]
  [make-snip make-snip]
  [horiz-margin 5] [vert-margin 5]
  [horizontal-inset 5] [vertical-inset 5])

frame)
```

7 Scribble Utilities

This library is *unstable*; compatibility will not be maintained. See *Unstable: May Change Without Warning* for more information.

```
(require unstable/gui/scribble)      package: unstable-lib
```

```
(codeblock->pict block) → pict?  
  block : block?
```

Converts a scribble block element into a pict.