Picturing Programs Teachpack

Version 9.4.0.5

Stephen Bloch

November 10, 2026

```
(require picturing-programs)
```

package: `picturing-programs`

1 About This Teachpack

Provides a variety of functions for combining and manipulating images and running interactive animations. It's intended to be used with the textbook Picturing Programs.

2 Installation

This package should be bundled with DrRacket version 5.1 and later, so there should be no installation procedure.

3 Functions from `2htdp/image` and `2htdp/universe`

This package includes all of `the image teachpack` and and `the universe teachpack`, so if you're using this teachpack, *don't* also load either of those. See the above links for how to use those teachpacks.

It also supersedes the older `tiles` and `sb-world` teachpacks, so if you have those, don't load them either; use this instead.

This package also provides the following additional functions:

4 Animation support

Since the Picturing Programs textbook introduces animations with image models before other model types, we provide a draw handler for the simple case in which the model is exactly what should be displayed in the animation window:

```
(show-it img) → image?
  img : image?
```

Returns the given image unaltered. Useful as a draw handler for animations whose model is an image.

5 New image functions

```
(rotate-cw img) → image?
  img : image?
```

Rotates an image 90 degrees clockwise.

```
(rotate-ccw img) → image?
  img : image?
```

Rotates an image 90 degrees counterclockwise.

```
(rotate-180 img) → image?
  img : image?
```

Rotates an image 180 degrees around its center.

```
(crop-top img pixels) → image?
  img : image?
  pixels : natural-number/c
```

Chops off the specified number of pixels from the top of the image.

```
(crop-bottom img pixels) → image?
  img : image?
  pixels : natural-number/c
```

Chops off the specified number of pixels from the bottom of the image.

```
(crop-left img pixels) → image?
  img : image?
  pixels : natural-number/c
```

Chops off the specified number of pixels from the left side of the image.

```
(crop-right img pixels) → image?
  img : image?
  pixels : natural-number/c
```

Chops off the specified number of pixels from the right side of the image.

```
(flip-main img) → image?
  img : image?
```

Reflects an image across the line x=y, moving the pixel at coordinates (x,y) to (y,x). The top-right corner becomes the bottom-left corner, and vice versa. Width and height are swapped.

```
(flip-other img) → image?
  img : image?
```

Reflects an image by moving the pixel at coordinates (x,y) to (h-y, w-x). The top-left corner becomes the bottom-right corner, and vice versa. Width and height are swapped.

```
(reflect-vert img) → image?
  img : image?
```

The same as `flip-vertical`; retained for compatibility.

```
(reflect-horiz img) → image?
  img : image?
```

The same as `flip-horizontal`; retained for compatibility.

```
(reflect-main-diag img) → image?
  img : image?
```

The same as `flip-main`; retained for compatibility.

```
(reflect-other-diag img) → image?
  img : image?
```

The same as `flip-other`; retained for compatibility.

6 Variables

This teachpack also defines variable names for some of the pictures used in the textbook.

`pic:bloch : image?`

A picture of the author, c. 2005.

`pic:hieroglyphics : image?`

A picture of a stone tablet with hieroglyphics on it.

`pic:hacker : image?`

A picture of a student sitting at a computer.

`pic:book : image?`

A picture of a book with a question mark.

`pic:stick-figure : image?`

A picture of a stick figure, built from geometric primitives.

`pic:scheme-logo : image?`

A picture of a DrScheme/DrRacket logo.

`pic:calendar : image?`

A picture of an appointment calendar.

Note that these seven variable names happen to start with "pic:", to distinguish them from anything you might define that happens to be named "calendar" or "book", but you can name a variable anything you want; in particular, there's no requirement that your names start with "pic:".

7 Pixel functions

The above functions allow you to operate on a picture as a whole, but sometimes you want to manipulate a picture pixel-by-pixel.

7.1 Colors and pixels

Each pixel of a bitmap image has a `color`, a built-in structure with four components – red, green, blue, and alpha – each represented by an integer from 0 to 255. Larger alpha values are "more opaque": an image with alpha=255 is completely opaque, and one with alpha=0 is completely transparent.

Even if you're not trying to get transparency effects, alpha is also used for dithering to smooth out jagged edges. In `(circle 50 "solid" "red")`, the pixels inside the circle are pure red, with alpha=255; the pixels outside the circle are transparent (alpha=0); and the pixels on the boundary are red with various alpha values (for example, if one quarter of a pixel's area is inside the mathematical boundary of the circle, that pixel's alpha value will be 63).

```
(name->color name) → (or/c color? false/c)
  name : (or/c string? symbol?)
```

Given a color name like "red", 'turquoise, "forest green", *etc.*, returns the corresponding color struct, showing the red, green, blue, and alpha components. If the name isn't recognized, returns `false`.

```
(colorize thing) → (or/c color? false/c)
  thing : (or/c color? string? symbol? false/c)
```

Similar to `name->color`, but accepts colors and `false` as well: colors produce themselves, while `false` produces a transparent color.

```
(color=? c1 c2) → boolean?
  c1 : (or/c color? string? symbol? false/c)
  c2 : (or/c color? string? symbol? false/c)
```

Compares two colors for equality. As with `colorize`, treats `false` as a transparent color (i.e. with an alpha-component of 0). All colors with alpha=0 are considered equal to one another, even if they have different red, green, or blue components.

```
(get-pixel-color x y pic) → color?
  x : natural-number/c
  y : natural-number/c
  pic : image?
```

Gets the color of a specified pixel in the given image. If x and/or y are outside the bounds of the image, returns a transparent color.

7.2 Specifying the color of each pixel of an image

```
(build-image width height f) → image?
  width : natural-number/c
  height : natural-number/c
  f : (-> natural-number/c natural-number/c color?)
```

Builds an image of the specified size and shape by calling the specified function on the coordinates of each pixel. For example,

```
; fuzz : image -> image
(define (fuzz pic)
  (local [; near-pixel : num(x) num(y) -> color
          (define (near-pixel x y)
            (get-pixel-color (+ x -3 (random 7))
                             (+ y -3 (random 7))
                             pic))]
    (build-image (image-width pic)
                 (image-height pic)
                 near-pixel)))
```

produces a fuzzy version of the given picture by replacing each pixel with a randomly chosen pixel near it.

```
(build-image/extra width height f extra) → image?
  width : natural-number/c
  height : natural-number/c
  f : (-> natural-number/c natural-number/c any/c color?)
  extra : any/c
```

Passes the *extra* argument in as a third argument in each call to *f*. This allows students who haven't learned closures yet to do pixel-by-pixel image manipulations inside a function depending on a parameter of that function.

For example, the above fuzz example could also be written as

```
; near-pixel : number(x) number(y) image -> color
(define (near-pixel x y pic)
  (get-pixel-color (+ x -3 (random 7))
                   (+ y -3 (random 7))
```

```
                  pic))
; fuzz : image -> image
(define (fuzz pic)
  (build-image/extra (image-width pic)
                     (image-height pic)
                     near-pixel
                     pic))
```

```
(build4-image width
              height
              red-function
              green-function
              blue-function
              alpha-function) → image?
  width : natural-number/c
  height : natural-number/c
  red-function : (-> natural-number/c natural-number/c natural-number/c)
  green-function : (-> natural-number/c natural-number/c natural-number/c)
  blue-function : (-> natural-number/c natural-number/c natural-number/c)
  alpha-function : (-> natural-number/c natural-number/c
                   natural-number/c)
```

A version of build-image for students who don't know about structs yet. Each of the four functions takes in the x and y coordinates of a pixel, and should return an integer from 0 through 255 to determine that color component.

```
(build3-image width
              height
              red-function
              green-function
              blue-function) → image?
  width : natural-number/c
  height : natural-number/c
  red-function : (-> natural-number/c natural-number/c natural-number/c)
  green-function : (-> natural-number/c natural-number/c natural-number/c)
  blue-function : (-> natural-number/c natural-number/c natural-number/c)
```

Just like build4-image, but without specifying the alpha component (which defaults to 255, fully opaque).

```
(map-image f img) → image?
  f : (-> color? color?)
  img : image?
(map-image f img) → image?
  f : (-> natural-number/c natural-number/c color?  color?)
  img : image?
```

Applies the given function to each pixel in a given image, producing a new image the same size and shape. The color of each pixel in the result is the result of calling f on the corresponding pixel in the input. If f accepts 3 parameters, it will be given the x and y coordinates and the color of the old pixel; if it accepts 1, it will be given only the color of the old pixel.

An example with a 1-parameter function:

```
; lose-red : color -> color
(define (lose-red old-color)
  (make-color 0 (color-green old-color) (color-blue old-color)))

(map-image lose-red my-picture)
```

produces a copy of `my-picture` with all the red leached out, leaving only the blue and green components.

Since `make-color` defaults alpha to 255, this definition of `lose-red` discards any alpha information (including edge-dithering) that was in the original image. To preserve this information, one could write

```
(define (lose-red-but-not-alpha old-color)
  (make-color 0 (color-green old-color) (color-blue old-
color) (color-alpha
old-color)))
```

An example with a 3-parameter (location-sensitive) function:

```
; apply-gradient : num(x) num(y) color -> color
(define (apply-gradient x y old-color)
  (make-color (min (* 3 x) 255)
              (color-green old-color)
              (color-blue old-color)))

(map-image apply-gradient my-picture)
```

produces a picture the size of `my-picture`'s bounding rectangle, replacing the red component with a smooth color gradient increasing from left to right, but with the green and blue components unchanged.

```
(map-image/extra f img extra) → image?
  f : (-> color? any/c color?)
  img : image?
  extra : any/c
(map-image/extra f img extra) → image?
  f : (-> natural-number/c natural-number/c color? any/c color?)
  img : image?
  extra : any/c
```

Passes the `extra` argument in as an additional argument in each call to `f`. This allows students who haven't learned closures yet to do pixel-by-pixel image manipulations inside a function depending on a parameter of that function.

For example,

```
; clip-color : color number -> color
(check-expect (clip-color (make-color 30 60 90) 100)
              (make-color 30 60 90))
(check-expect (clip-color (make-color 30 60 90) 50)
              (make-color 30 50 50))
(define (clip-color c limit)
    (make-color (min limit (color-red c))
                (min limit (color-green c))
                (min limit (color-blue c))))

; clip-picture-colors : number(limit) image -> image
(define (clip-picture-colors limit pic)
    (map-image/extra clip-color pic limit))
```

This `clip-picture-colors` function clips each of the color components at most to the specified limit.

Another example, using x and y coordinates as well:

```
; new-pixel : number(x) number(y) color height -> color
(check-expect (new-pixel 36 100 (make-color 30 60 90) 100)
              (make-color 30 60 255))
(check-expect (new-pixel 58 40 (make-color 30 60 90) 100)
              (make-color 30 60 102))
(define (new-pixel x y c h)
  (make-color (color-red c)
              (color-green c)
              (real->int (* 255 (/ y h)))))

; apply-blue-gradient : image -> image
(define (apply-blue-gradient pic)
  (map-image/extra new-pixel pic (image-height pic)))
```

This `apply-blue-gradient` function changes the blue component of an image to increase gradually from the top to the bottom of the image, (almost) reaching 255 at the bottom of the image.

```
(map4-image red-func
            green-func
            blue-func
            alpha-func
            img)       → image?
  red-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natural-
  green-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natura
  blue-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natural
  alpha-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natura
  img : image?
```

A version of map-image for students who don't know about structs yet. Each of the four given functions is assumed to have the contract

```
num(x) num(y) num(r) num(g) num(b) num(alpha) -> num
```

For each pixel in the original picture, applies the four functions to the x coordinate, y coordinate, red, green, blue, and alpha components of the pixel. The results of the four functions are used as the red, green, blue, and alpha components in the corresponding pixel of the resulting picture.

For example,

```
; each function : num(x) num(y) num(r) num(g) num(b) num(a) -> num
(define (zero x y r g b a) 0)
(define (same-g x y r g b a) g)
(define (same-b x y r g b a) b)
(define (same-alpha x y r g b a) a)
(map4-image zero same-g same-b same-alpha my-picture)
```

produces a copy of `my-picture` with all the red leached out, leaving only the blue, green, and alpha components.

```
; each function : num(x) num(y) num(r) num(g) num(b) num(a) -> num
(define (3x x y r g b a) (min (* 3 x) 255))
(define (3y x y r g b a) (min (* 3 y) 255))
(define (return-255 x y r g b a) 255)
(map4-image 3x zero 3y return-255 my-picture)
```

produces an opaque picture the size of `my-picture`'s bounding rectangle, with a smooth color gradient with red increasing from left to right and blue increasing from top to bottom.

```
(map3-image red-func
            green-func
            blue-func
            img)       → image?
```

```
  red-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natural-
  green-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natura
  blue-func : (-> natural-number/c natural-number/c natural-number/c natural-number/c natural
  img : image?
```

Like map4-image, but not specifying the alpha component. Note that the red, green, and blue functions also *don't take in* alpha values. Each of the three given functions is assumed to have the contract

```
num(x) num(y) num(r) num(g) num(b) -> num
```

For each pixel in the original picture, applies the three functions to the x coordinate, y coordinate, red, green, and blue components of the pixel. The results are used as a the red, green, and blue components in the corresponding pixel of the resulting picture.

The alpha component in the resulting picture is copied from the source picture. For example,

```
; each function : num(x) num(y) num(r) num(g) num(b) -> num
(define (zero x y r g b) 0)
(define (same-g x y r g b) g)
(define (same-b x y r g b) b)
(map3-image zero same-g same-b my-picture)
```

produces a copy of my-picture with all the red leached out; parts of the picture that were transparent are still transparent, and parts that were dithered are still dithered.

```
; each function : num(x) num(y) num(r) num(g) num(b) num(a) -> num
(define (3x x y r g b a) (min (* 3 x) 255))
(define (3y x y r g b a) (min (* 3 y) 255))
(map3-image zero 3x 3y my-picture)
```

produces a my-picture-shaped "window" on a color-gradient.

```
(fold-image f init img) → any/c
  f : (-> color? any/c any/c)
  init : any/c
  img : image?
(fold-image f init img) → any/c
  f : (-> natural-number/c natural-number/c color? any/c any/c)
  init : any/c
  img : image?
```

Summarizes information from all the pixels of an image. The result is computed by applying f successively to each pixel, starting with *init*. If *f* accepts four parameters, it is called with

the coordinates and color of each pixel as well as the previously-accumulated result; if it accepts two parameters, it is given just the color of each pixel and the previously-accumulated result. You may not assume anything about the order in which the pixels are visited, only that each pixel will be visited exactly once.

An example with a 2-parameter function:

```
; another-white : color number -> number
(define (another-white c old-total)
   (+ old-total (if (color=? c "white") 1 0)))

; count-white-pixels : image -> number
(define (count-white-pixels pic)
   (fold-image another-white 0 pic))
```

Note that the accumulator isn't restricted to be a number: it could be a structure or a list, enabling you to compute the average color, or a histogram of colors, etc.

```
(fold-image/extra f init img extra) → any/c
  f : (-> color? any/c any/c any/c)
  init : any/c
  img : image?
  extra : any/c
(fold-image/extra f init img extra) → any/c
  f : (-> natural-number/c natural-number/c color? any/c any/c any/c)
  init : any/c
  img : image?
  extra : any/c
```

Like `fold-image`, but passes the *extra* argument in as an additional argument in each call to *f*. This allows students who haven't learned closures yet to call `fold-image` on an operation that depends on a parameter to a containing function.

For example,

```
; another-of-color : color number color -> number
(define (another-of-color c old color-to-count)
   (+ old (if (color=? c color-to-count) 1 0)))

; count-pixels-of-color : image color -> number
(define (count-pixels-of-color pic color-to-count)
   (fold-image/extra another-of-color 0 pic color-to-count))
```

```
(real->int num) → integer?
  num : real?
```

Not specific to colors, but useful if you're building colors by arithmetic. For example,

```
; bad-gradient : num(x) num(y) -> color
(define (bad-gradient x y)
  (make-color (* 2.5 x) (* 1.6 y) 0))
(build-image 50 30 bad-gradient)

; good-gradient : num(x) num(y) -> color
(define (good-gradient x y)
  (make-color (real->int (* 2.5 x)) (real->int (* 1.6 y)) 0))
(build-image 50 30 good-gradient)
```

The version using `bad-gradient` crashes because color components must be exact integers. The version using `good-gradient` works.

8 Input and Output

This teachpack also provides several functions to help in testing I/O functions (in Advanced Student language; ignore this section if you're in a Beginner or Intermediate language):

```
(with-input-from-string input thunk) → any/c
  input : string?
  thunk : (-> any/c)
```

Calls `thunk`, which presumably uses `read`, in such a way that `read` reads from `input` rather than from the keyboard.

```
(with-output-to-string thunk) → string?
  thunk : (-> any/c)
```

Calls `thunk`, which presumably uses `display`, `print`, `write`, and/or `printf`, in such a way that its output is accumlated into a string, which is then returned.

```
(with-input-from-file filename thunk) → any/c
  filename : string?
  thunk : (-> any/c)
```

Calls `thunk`, which presumably uses `read`, in such a way that `read` reads from the specified file rather than from the keyboard.

```
(with-output-to-file filename thunk) → any/c
  filename : string?
  thunk : (-> any/c)
```

Calls `thunk`, which presumably uses `display`, `print`, `write`, and/or `printf`, in such a way that its output is redirected into the specified file.

```
(with-input-from-url url thunk) → any/c
  url : string?
  thunk : (-> any/c)
```

Calls `thunk`, which presumably uses `read`, in such a way that `read` reads from the HTML source of the Web page at the specified URL rather than from the keyboard.

```
(with-io-strings input thunk) → string?
  input : string?
  thunk : (-> any/c)
```

Combines `with-input-from-string` and `with-output-to-string`: calls `thunk` with its input coming from `input` and accumulates its output into a string, which is returned. Especially useful for testing:

```
; ask : string -> prints output, waits for text input, returns it
(define (ask question)
  (begin (display question)
         (read)))
; greet : nothing -> prints output, waits for text input, prints output
(define (greet)
  (local [(define name (ask "What is your name?"))]
    (printf "Hello, ~a!" name)))
(check-expect
  (with-io-strings "Steve" greet)
  "What is your name?Hello, Steve!")
```