Processing single images

rotate-cw: image -> image  [also rotate-ccw, rotate-180]
Return the image, rotated clockwise 90 degrees

rotate: number(degrees) image -> image
Return the image, rotated by the specified number of degrees

scale: number image -> image
Return the image, scaled by separate width and height factors

scale/xy: number(width) number(height) image -> image
Return the image, scaled by specified number

breakpoint: image -> image
Return the image, flipped top to bottom

撮影: image -> image
Return a framed image

image-height: image -> number [also image-width]
Return the height of an image

image?: any -> Boolean
Return true if the input is an image

image=?: image image -> Boolean
Determine whether 2 images are equal

build-image: number(width) number(height) function(num(x) num(y) -> color) -> image
Build an image of the specified size, applying the function at every (x,y) posn

map-image: function(num(x) num(y) -> color) image image image image ...
Apply the function at each x-y posn of the original image to create new image

save-image: image string(file-name) -> Boolean
Writes image to file name/path specified by the string (PNG format, so “whatever.png”)

bitmap: image string(file-name) -> image
Read the image from the specified file name/path, returning the image

Combining images

above: image image ... -> image
Stack the input images vertically, with the first image on top

above/align: string(right/left/middle) image image ... -> image
Stack the input images vertically, first image on top, aligned as specified

beside: image image ... -> image
Place the input images next to each other horizontally, first image on the left

beside/align: string(top/bottom/middle) image image ... -> image
Place the input images next to each other horizontally, first image on the left, aligned as specified
; add-line: image number(x) number(y) number(a) number(b) -> image
; Add a line from (x,y) to (a,b) into an image

; overlay: image image ... -> image
; Return the first input image on top of the second, both on top of the third, ...

; overlay/align: string(left/right/middle) string(top/bottom/middle) image image ... -> image
; Return stacked images like overlay, but aligned horizontally and vertically as specified

; overlay/xy: image number(x) number(y) image -> image
; Overlay the first image on top of the second image after shifting the second x right and y down

[Also underlay, underlay/align, underlay/xy]

; place-image: image number(x) number(y) image -> image
; Place the first image with its center at (x,y) into the second image, cropping to maintain the
; boundaries of the second image

Posns and colors

; make-posn: number number -> posn
; Construct a posn

; posn-x: posn -> number
; Extract the x-component of the posn

; posn?: anything -> Boolean
; Determine if a value is a posn

; color-red: color -> number
; Extract the red value from a color

; color?: anything -> Boolean
; Determine whether a value is a color

; make-color: number(red) number(green) number(blue) -> color
; Create a color

Design recipe in a nutshell
1. Determine what type(s) of data will be input and what type will be returned.
2. Contract. 3. Purpose statement. 4. Examples (check-expect …) 5. Function header

Scheme functions, constants, and special forms you do need to know
Note that the important thing is how to use these, what the common usage patterns are. You should
study whole functions and programs, not this list of functions in isolation.

Numbers: + - * / add1 sub1 < <= -> >

Strings: "" string=? string<? string<=? string>? string>=?

Booleans and predicates: true false and or not equal?

Structures: define-struct and structure operations (constructor, accessors, checker)
Lists: empty? cons first rest empty? cons? length member
list append list-ref
Lists (higher-order functions): build-list map filter foldr quicksort
Control: cond else
I/O and other imperative features: read read-line display newline begin set!
Vectors: vector vector-ref vector-set! build-vector
Testing: check-expect check-within check-range check-member-of

[Compiled by Martina Mickos and David G. Kay. Errors or suggestions to kay@uci.edu.]