Scheme Reference Sheet for Informatics 41 and *Picturing Programs*

[v1.0]

It is not important that you memorize the names and behavior of all the image processing and animation functions we might use. This sheet provides a compact reference. Use it in the lab; we will also provide you a copy on quizzes when it's necessary. If you encounter an error or a necessary function that isn't listed here, let us know. (There are many more functions available; check the 2htdp/image and 2htdp/universe teachpacks in the DrRacket help desk.)

Creating shapes

; **circle**: number(radius) string(outline/solid) string(color) -> image
; Create a circle with specified radius, mode, and color

; **ellipse**: number(width) number(height) string(outline/solid) string(color) -> image
; Create an ellipse with specified width, height, mode, and color

; **line**: number(x) number(y) -> image
; Create a line that connects (0,0) to (x,y)

; **square**: number(side) string(outline/solid) string(color) -> image
; Create a square with specified side length, mode, and color

; **rectangle**: number(width) number(height) string(solid/outline) string(color) -> image
; Create a rectangle with specified width, height, mode, and color

; **rhombus**: number(side) number(angle) string(outline/solid) string(color) -> image
; Create a rhombus with specified side length, side angle, mode, and color

; **triangle**: number(side) string(solid/outline) string(color) -> image
; Create an equilateral triangle with specified side length, mode, and color

; **right-triangle**: number(side) number(side) string(outline/solid) string(color) -> image
; Create a right triangle with specified side lengths, mode, and color

; **isosceles-triangle**: number(side) number(angle) string(outline/solid) string(color) -> image
; Create an isosceles triangle with specified side length, angle, mode, and color

; **star**: number(side) string(outline/solid) string(color) -> image
; Create a star with specified side length, mode, and color

; **polygon**: number(vertices) string(outline/solid) string(color) -> image
; Create a polygon with specified number of vertices, mode, and color

; **text**: string(text) number(font-size) string(color) -> image
; Constructs an image with the specified text in the specified size and color

; **text/font**: string(text) number(font-size) string(color) string(face) symbol(family)
; symbol(normal/italic/slant) symbol(normal/bold/light) string(#t/#f) -> image
; Constructs an image with the specified text in the specified size, color, typeface,
; type family (e.g. 'roman or 'script) in case the specified typeface isn't available,
; style, weight, and underlining status (#t for yes, #f for no)

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 specified number

; **scale/xy**: number(width) number(height) image -> image
; Return the image, scaled by separate width and height factors

; **crop-left**: image number -> image [also **crop-right**, **crop-top**, **crop-bottom**]
; Return the image with the specified number of pixels removed from the left

; **crop**: number(x) number(y) number(width) number(height) image -> image
; Return a cropped image with upper left point (x,y), height and width specified

; **flip-vertical**: image -> image [also **flip-horizontal**]
; Return the image, flipped top to bottom

; **frame**: 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 -> color) 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 [also **posn-y**]
; Extract the x-component of the posn

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

; **color-red**: color -> number [also **color-green**, **color-blue**]
; 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

Animations/Worlds

To set up an animation, you need to decide what data you need to model your world and set up a call to big-bang, specifying the initial world and the callback functions for various events. In the example below, the bold text indicates required keywords; normal text indicates programmer-chosen names.

```
(big-bang
  initial-world                ; The starting value for your world
  (check-with world-checker?)  ; Specify function to check at each tick that the world is the right type
  (to-draw draw-handler)       ; Specify function to call to redraw the world at each event
  (on-tick tick-handler 1)     ; Specify function to call at each tick (here, 1 sec.), to update the world
  (on-key key-handler)         ; Specify function to call each time the user hits a key
  (on-mouse mouse-handler)     ; Specify function to call each time the user clicks or moves the mouse
  (stop-when end-checker?))    ; Specify function to call at each tick to check whether to quit
```

; **empty-scene**: number(width) number(height) -> image
; Return an empty scene of the specified size, useful for building the initial world

; **show-it**: image -> image
; Return the image unaltered; useful as a draw-handler when the display doesn't change

; **stop-with**: world(current) -> world(current) with a signal to big-bang to redraw once and quit.
; Called within a handler, this stops the animation after redrawing the world one last time.

The event handler (callback) functions have the contracts shown below. Their names are identifiers; you can choose your own names. But their contracts indicate what values each function gives you to work with and what type of expression each function needs to return.

; world-checker?: any -> Boolean
; This function checks at each tick that the world is still the right type of data; it helps snag errors.

; draw-handler: world(current) -> image
; This function can look at the current world and produce an image (that will be the new screen)

; tick-handler: world(current) -> world(new)
; This function can look at the current world as it decides how to update the world for the next tick

; key-handler: world(current) string(key-event) -> world(new)
; Key events include "shift" "control" "up" [arrow] "\r" [return] and one-character key names like "a"
; This function updates the world any time a key is pressed; it looks at the world and which key it was.

; mouse-handler: world(current) number(x) number(y) string(mouse-event) -> world(new)
; Mouse events include "button-down" "button-up" "drag" "move" "enter" "leave"
; This function updates the world any time a mouse event occurs; it receives the (x,y) location
; where the mouse event occurred.

; end-checker?: world -> Boolean
; This function looks at the world and decides if the animation should be over (returning true if so)

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
6. Inventory and template as needed. 7. Body of function. 8. Run tests.

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 < <= = >= >`
`zero? random min max odd? even?`
Strings: `""` `string=? string<? string<=? string>? string>=?`
`string-length`
Booleans and predicates: `true false and or not equal?`
`number? string?`
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.]