Solutions for Sample Midterm Exam #1

We use the design recipe and given the data representation chosen we rely on the function template for atomic input data for this problem.

```
; Problem 1.
; Temperature is a
; -- Number
; Number -> Number
; converts Fahrenheit value to its Celsius equivalent
; examples: given 98 expected 36.6
; given 99.23 expected 37.35
; given 32 expected 0
; (define (f->c input)
; ( ... input ... )
(define (f->c input)
  (* (/ 5 9) (- input 32))))
(check-expect (f->c 98) 36.6)
(check-expect (f->c 99.23) 37.35)
(check-expect (f->c 32) 0)
```

We always use the design recipe and in this problem given the data representation chosen we rely on the function template for recursive input data.

```
; Problem 2.1
; Temperature is a
; -- Number
; ManyTemperatures is one of:
; -- Temperature
; -- (make-many Temperature ManyTemperatures)
(define-struct many (temp temps))
; some examples
(define ex1 98)
(define ex2 (make-many 99.23 ex1))
(define ex3 (make-many 32 ex2))
(define ex4 (make-many 97.2 (make-many 97.81 99.01)))
; ManyTemperatures -> Number
; sums up the temperatures in a ManyTemperatures
; examples: given ex1 expected 98
;            ex2 expected 197.23
;            ex3 expected 229.23
; (define (sum input)
;  (cond ((number? input) ...)
;        (else ( ... (many-temp input) ... (sum (many-temps input)) ... ))))
(define (sum input)
  (cond ((number? input) input)
       (else (+ (many-temp input) (sum (many-temps input))))))
(check-expect (sum ex1) 98)
(check-expect (sum ex2) 197.23)
(check-expect (sum ex3) 229.23)
```
The third problem is almost identical with the second:

```scheme
; Problem 2.2
; ManyTemperatures -> Number
; counts the temperatures in a ManyTemperatures
; examples: given ex1 expected 1
; ex2 expected 2
; ex3 expected 3
; (define (count input)
;   (cond ((number? input) ...)
;     (else ( ... (many-temp input) ... (count (many-temps input)) ... ))))
(define (count input)
  (cond ((number? input) 1)
    (else (+ 1 (count (many-temps input))))))
(check-expect (count ex1) 1)
(check-expect (count ex2) 2)
(check-expect (count ex3) 3)
```

For the third problem we notice we can use the previous two functions as helpers and the template for functional (de)composition:

```scheme
; Problem 2.3
;
; sum: ManyTemperatures -> Number
; count: ManyTemperatures -> Number
; average: Number Number -> Number
;
; ManyTemperatures -> Number Number -> Number
; examples given ex1 expected 98
; ex2 expected 98.615
; ex3 expected 76.41
; (define (average input)
;   (cond ( ... (sum input) ... (count input) ... ) )))
(define (average input)
  (/ (sum input) (count input)))
(check-expect (average ex1) 98)
(check-expect (average ex2) 98.615)
(check-expect (average ex3) 76.41)
```

Notice that even if we use the recursive input data function template we still need a helper:

```scheme
; Problem 2.3 (using recursive input data function template)
;
; ManyTemperatures -> Number
; same examples as before
; (define (avg input)
;   (cond ((number? input) ...)
;     (else ( ... (many-temp input) ... (avg (many-temps input)) ... ) )))
(define (avg input)
  (cond ((number? input) input)
    (else ( / (+ (many-temp input)
        (* (count (many-temps input))
        (avg (many-temps input))))
      (count input))))
(check-expect (avg ex1) 98)
(check-expect (avg ex2) 98.615)
(check-expect (avg ex3) 76.41)
```
For the next problem we choose a data representation based on lists that allows the empty list in our input. The function template is a combination of the recursive input data template and the functional (de)composition using the $f\rightarrow c$ function defined earlier:

```
; problem 3
;
; Temperature is a
; -- Number
; ListOfTemperatures is one of:
; -- empty
; -- (cons Temperature ListOfTemperatures)
;
; ListOfTemperatures -> ListOfTemperatures
;
; examples given ex1 expected
; ex2 expected
; ex3 expected
;
; (define (convert inputs)
;  (cond ((empty? inputs) ...)
;     (else (... (first inputs) ... (convert (rest inputs)) ...) )))
(define (convert inputs)
  (cond ((empty? inputs) empty)
     (else (cons (f->c (first inputs)) (convert (rest inputs))))))
)
(check-expect (convert empty) empty)
(check-expect (convert (list 98.62)) (list 37.01))
(check-expect (convert (list 99.01 97.82)) (list 37.227 36.56))
(check-expect (convert (list 96.99 97.37 98.21)) (list 36.105 36.316 36.783))
```

For the last problem we start from the big-bang template and define our chosen data representation for the World and how the world changes as time passes:

```
(require 2htdp/image)
(require 2htdp/universe)

; (define (main initial-world)
;  (big-bang initial-world
;     (on-tick change-world ...)
;     (to-draw render-world)))

; A World is a
; -- Number

(define initial 98)

; World -> World
; changes the world as time flies by
; (define (change-world world)
;  (... world ...)
(define (change-world world)
  (+ world (/ (- (random 40) 20) 100)))
```

Notice that in the process we also establish the name of the function that renders the world.
Its development follows the design recipe and uses an itemization function template combined with functional (de)composition:

```
; Number -> Image
; renders the world as a traffic light
(define (render-world world)
  (cond ((>= world 100.9) ...)
    ((> 100.9 world 99.5) ...)
    ((>= 99.5 world 95.9) ...)
    ((> 95.9 world) ...)))

(define (render-world world)
  (cond ((> world 100.9) (show-text world "red"))
    ((> 100.9 world 99.5) (show-text world "yellow"))
    ((>= 99.5 world 95.9) (show-text world "green"))
    ((> 95.9 world) (show-text world "blue"))))

; Number Color -> Image
(define (show-text world color)
  (place-image
    (text (number->string (exact->inexact world)) 34 color)
    60 35
    (empty-scene 120 70)))

(define (main initial-world)
  (big-bang initial-world
    (on-tick change-world 1)
    (to-draw render-world)))
```

Here’s what this program looks like in its various stages:

In case you’re wondering this solution should definitely receive full credit since it accomplishes reasonably the initial goal. However trying to work out a solution where the individual lights show as well is very educational and should be encouraged (now, as we prepare for the exam,
or during the exam). Here’s what the render function becomes if we want to achieve the original specifications:

```scheme
; Number -> Image
; renders the world as a traffic light
; uses functional decomposition
(define (render-world world)
  (beside (show-text world)
    (show-blue world)
    (show-green world)
    (show-yellow world)
    (show-red world)))

; Number Color -> Image
(define (show-text world)
  (text (number->string (exact->inexact world)) 34 "black"))

(define (show-red world)
  (cond ((>= world 100.9) (circle 30 "solid" "red"))
         (else (circle 30 "outline" "red")))

(define (show-yellow world)
  (cond ((> 100.9 world 99.5) (circle 30 "solid" "yellow"))
         (else (circle 30 "outline" "yellow")))

(define (show-green world)
  (cond ((> 99.5 world 95.9) (circle 30 "solid" "green"))
         (else (circle 30 "outline" "green")))

(define (show-blue world)
  (cond ((> 95.9 world) (circle 30 "solid" "blue"))
         (else (circle 30 "outline" "blue")))

(define (main initial-world)
  (big-bang initial-world
    (on-tick change-world 1)
    (to-draw render-world)))

And a reminder how this function renders the world (four different cases):

![Four different cases of rendered world with different colors based on the world value.](image)

Note: two problems have been solved twice in this document, that’s why it is so long.