CS 173 Programming Languages Shriram K.

Scheme Tutorial Solutions

Fall 2002

Problem Set 3: Higher-order functions

21. Specific *check-range* function for determining whether a list of temperatures is between $5°$C and $95°$C inclusive. This function uses the *foldl* function.

```
;; check-range1 : (listof number) → boolean
;; determines if all numbers in a list are between 5 and 95
(define (check-range1 temp-list)
  (foldl (lambda (temp accum) (and accum (<= 5 temp) (<= temp 95))) true temp-list))
```

General *check-range* function for determining whether a list of temperatures is between a min and a max value.

```
;; check-range : (listof number) number number → boolean
;; determines if all numbers in a list are between a specified max and min value
(define (check-range temp-list min max)
  (foldl (lambda (temp accum) (and accum (<= min temp) (<= temp max))) true temp-list))
```

22. Function which converts a list of digits to the corespodning number, using the *foldr* function.

```
;; convert : (listof number) → number
;; converts a list of digits to the coresponding number
(define (convert list-digits)
  (foldr (lambda (digit accum) (+ (* 10 accum) digit)) 0 list-digits))
```

23. Functions which together compute the average of a list of prices

```
;; count : (listof number) → number
(define (count lotp)
  (foldl (lambda (elem accum) (+ 1 accum)) 0 lotp))

;; sum : (listof number) → number
(define (sum lotp)
  (foldl (lambda (elem accum) (+ elem accum)) 0 lotp))
```

```
;; average-price : (listof number) → number
;; finds the average value of a list of numbers
(define (average-price lotp)
  (/ (sum lotp) (count lotp)))
```

24. Function for converting a list of Farenheit tempuratures to a list of Celsius temperatures using the *map* function.

```
;; tempFC: (listof number) → (listof number)
;; Purpose: To convert a list of Farenheit temps to a list of Celsius temps
(define (tempFC list-Ftemps)
  (map (lambda (tempF) (* (/ 5 9) (- tempF 32))) list-Ftemps))
```

25. Function which uses the *filter* function to remove all toys with a price greater than *ua* from a list of toy prices.

```
;; eliminate-exp: (listof number) number → (listof number)
;; removes all values above a user specified value from a list
(define (eliminate-exp lotp ua)
  (filter (lambda (x) (<= x ua)) lotp))
```

26. Function which creates a function representing the composition of two functions.

```
;; compose-func: (Y → Z) (X → Y) → (X → Z)
;; creates a function which is the composition of two other functions
(define (compose-func a b)
  (lambda (x) (a (b x))))
```

27. Two versions, one using *foldr* and one without, of a function to convert a list of sublists of numbers to a list of numbers.

```
;; flatten : (listof (listof number)) → (listof number)
;; flattens a list of sublists of numbers into a list of numbers
(define (flatten list-of-lists)
  (cond
    [(empty? list-of-lists) empty]
    [else (append (first list-of-lists) (flatten (rest list-of-lists)))]))

;; flatten-foldr: (listof (listof number)) → (listof number)
(define (flatten-foldr list-of-lists)
  (foldr append empty list-of-lists))
```

28. Function using *foldr* to divide a list into sublists which are composed of adjacent equal numbers.

```
;; bucket : (listof number) → (listof (listof number))
;; divides a list into a list of sublists, where the sublists are
;; composed of adjacent equal numbers in the original list.
(define (bucket alon)
  (foldr (lambda (elem accum)
           (cond
             [(or (empty? accum)
                  (not (= (first (first accum)) elem)))
              (cons (cons elem empty) accum)]
             [else (cons (cons elem (first accum)) (rest accum))]))
         empty alon))
```

29. Function which applies a function *f* to the name of every person in the *family-tree*

```
;; tree-map: family-tree (string → string) → family-tree
;; applies the given function to the name of every person in a family tree
(define (tree-map f tree)
  (cases family-tree tree
    [unknown () (unknown)]
    [person (name birth eye mom dad) (person (f name) birth eye
                                             (tree-map f mom)
                                             (tree-map f dad))]))
```

30. Function which uses *tree-map* to add a last name to every person in a *family-tree*.

```
;; add-last-name : family-tree string → family-tree
;; appends a last name to the name of every person in a family tree
(define (add-last-name tree last-name)
  (tree-map (lambda (name) (string-append name " " last-name)) tree))
```