Abstract List Functions

;; build-list: Nat (Nat -> X) -> (listof X)
(build-list n f) produces (list (f 0) ... (f (sub1 n)))

;; filter: (X -> Bool) (listof X) -> (listof X)
(filter pred? lst) produces a list containing the elements of lst for which pred? holds.
Ordering of elements is preserved.

;; quicksort: (listof X) (X X -> Bool) -> (listof X)
(quicksort alox cmp) produces the items of alox sorted in order according to cmp

;; map: (X -> Y) (listof X) -> (listof Y)
(map f lst) produces a list by applying f to each element of lst.
That is, (map f (list x1 ... xn)) produces (list (f x1) ... (f xn))

;; foldr: (X Y -> Y) Y (listof X) -> Y
(foldr combine base lst) produces (combine x1 ... (combine xn base))
given that lst is (list x1 ... xn)

;; foldl: (X Y -> Y) Y (listof X) -> Y
(foldl combine base lst) produces (combine xn ... (combine x1 base))
given that lst is (list x1 ... xn)

Selected String and Character Functions

(string-length s) produces the number of characters in string s

(string-append s1 ... sn) produces a string formed by concatenating s1 ... sn

(string-ref s n) produces the character at position n in the string s
requires: 0 ≤ n < (string-length s)

(substring s p1) produces the string containing all characters from position p1 to end of s
requires: p1 ≤ (string-length s)

(substring s p1 p2) produces the string containing all characters from position p1 to
(- p2 1), where the first character is at position 0
requires: p1 ≤ p2 ≤ (string-length s)

Selected Other Functions

(quotient n m) produces the quotient when n is divided by m

(remainder n m) produces the remainder when n is divided by m

(positive? x) produces true if x > 0, false otherwise

(negative? x) produces true if x < 0, false otherwise

(even? x) produces true if x is even, false otherwise

(odd? x) produces true if x is odd, false otherwise

(zero? x) produces true if x is 0, false otherwise

(append lst1 ... lstn) produces a list of all elements in lst1 ... lstn, in that order

(length lst) produces the number of elements in lst

(member? x lst) produces true if x is in lst, false otherwise
Graphs

These are functions for finding a route in a graph that may contain cycles. The functions `neighbours`, `find-route/list` and `find-route/acc` match the code on Slide 12, Slide 31 and Slide 33 of Module 12 exactly. The function `find-route` is a wrapper function that uses `find-route/acc` to find a route from an origin node to a destination node in a graph.

```scheme
;; A Node is a Sym
;; A Graph is a (listof (list Node (listof Node)))

;; (neighbours v G) produces the list of neighbours of v in G
;; neighbours: Node Graph -> (listof Node)
;; requires: v is a node in G
(define (neighbours v G)
  (cond [(symbol=? v (first (first G))) (second (first G))]
        [else (neighbours v (rest G))]))

;; (find-route/acc orig dest G visited) produces a path of nodes from
;; orig to dest in G, or false if no such path exists.
;; find-route/acc: Node Node Graph (listof Node) -> (anyof (listof Node) false)
(define (find-route/acc orig dest G visited)
  (cond [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                     (define route (find-route/list nbrs dest G (cons orig visited)))]
                  (cond [(false? route) false]
                        [else (cons orig route)]))]))

;; (find-route/list los dest G visited) produces a path of nodes from a
;; member of los to dest in G if one exists, or false if there is no such path.
;; find-route/list: (listof Node) Node Graph (listof Node) -> (anyof (listof Node) false)
(define (find-route/list los dest G visited)
  (cond [(empty? los) false]
        [(member? (first los) visited)
         (find-route/list (rest los) dest G visited)]
        [else (local [(define route (find-route/acc (first los) dest G visited))]
                  (cond [(false? route) (find-route/list (rest los) dest G visited)]
                        [else route]))]))

;; (find-route orig dest G) produces a path from orig to dest in G if one
;; exists, or false otherwise.
;; find-route: Node Node Graph -> (anyof (listof Node) false)
(define (find-route orig dest G)
  (find-route/acc orig dest G empty))
```