Note: These are the abstract list functions taught in class. Racket has others, including other versions of map and foldr, but you are not allowed to use them.

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

;; (filter pred? lst) produces a list containing the elements of lst for which pred? holds
;; (the ordering of the elements from lst is preserved)
;; filter: (X → Bool) (listof X) → (listof X)
(define (filter pred? lst) ...)

;; (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))
;; map: (X → Y) (listof X) → (listof Y)
(define (map f lst) ...)

;; (foldr f base lst) produces (f x1 ... (f xn base)) given that lst = (list x1 ... xn)
;; foldr: (X Y → Y) Y (listof X) → Y
(define (foldr f base lst) ...)

;; (foldl f base lst) produces (f xn ... (f x1 base)) given that lst = (list x1 ... xn)
;; foldl: (X Y → Y) Y (listof X) → Y
(define (foldl f base lst) ...)
Graphs

;; (neighbours v G) produces list of neighbours of v in G
;; neighbours: Node Graph → (listof Node)
;; requires: v is in G
(define (neighbours v G)
  (cond [(empty? G) (error "vertex not in graph")]
          [(symbol=? v (first (first G))) (second (first G))]
          [else (neighbours v (rest G))]))

;; (find-route/list los dest G) produces route from
;; an element of los to dest in G, if one exists
;; find-route/list: (listof Node) Node Graph → (anyof (listof Node) false)
(define (find-route/list los dest G)
  (cond [(empty? los) false]
          [else (local [(define route (find-route (first los) dest G))]
                    (cond [(false? route) (find-route/list (rest los) dest G)]
                          [else route]))]))

;; (find-route orig dest G) finds route from orig to dest in G if it exists
;; find-route: Node Node Graph → (anyof (listof Node) false)
(define (find-route orig dest G)
  (cond [(symbol=? orig dest) (list orig)]
          [else (local [(define nbrs (neighbours orig G))
                        (define route (find-route/list nbrs dest G))]
                        (cond [(false? route) route]
                              [else (cons orig route)]))]))