Racket Lists

Creating lists in Racket

empty constructs an empty list

(list <value1> ... <valueN>) constructs a list with the given arguments as elements

(cons <value> <list>) given an element and another list, constructs a new list by prepending the element to the provided list. (Careful! If the second element is not a list, then cons does not construct a new list.)

Accessing lists in Racket

(empty? <value>) returns true if the argument is an empty list.

(first <list>) returns the first element of the list

(rest <list>) returns a list that contains every element except the first element of the list.

Inductive data types & recursive operations

Induction is a “building up” process: An inductive data type is one that can be built by starting with the smallest possible instance of that data type, then incrementally building larger instances of it. Racket lists (which are linked lists) are an example of an inductive data type.

Recursion is a “breaking down” process: A recursive function is one breaks a larger problem into a smaller version of that problem that is easier to solve.

If we define a recursive function over an inductive data type, our code usually follows a pattern. For example, here is a recursive function to compute the length of a Racket list:

(define (len L)
  (if (empty? L)          ; base case: is this the smallest list?
      0                ; If so, our job is easy.
      (+ 1 (len (rest L))))) ; recursive case: if it’s not the smallest
                         ; list, solve the problem on a smaller
                         ; list and use the results to compute
                         ; the size of this list.

Next time: functions as data!
(define (isEven N)
    (check-equal? (isEven -1) false)
    (check-equal? (isEven 0) true)
    (check-equal? (isEven 1) false)
    (check-equal? (isEven 2) true)
)

(define (fact N)
)

(define (fib N)
    (check-equal? (isEven -1) false)
    (check-equal? (isEven 0) true)
    (check-equal? (isEven 1) false)
    (check-equal? (isEven 2) true)
)

(define (halve-count: the computer scientist's log2 function)
    (check-equal? (isEven -1) false)
    (check-equal? (isEven 0) true)
    (check-equal? (isEven 1) false)
    (check-equal? (isEven 2) true)
)
;; sum:
;;   inputs: a list L of numbers
;;   outputs: the sum of the numbers in the list
(define (sum L)

; tests
(check-equal? (sum '()) 0)
(check-equal? (sum '(1)) 1)
(check-equal? (sum '(1 2 1)) 4)
(check-equal? (sum '(0 0 121 0)) 121)
(check-equal? (sum '(1 2 3 4)) 10)

;; product:
;;   inputs: a list L of numbers
;;   outputs: the product of the numbers in the list
(define (product L)

; tests
(check-equal? (product '()) 1)
(check-equal? (product '(1)) 1)
(check-equal? (product '(1 2 3)) 6)
(check-equal? (product '(1 2 121 0)) 0)
(check-equal? (product '(1 2 3 4)) 24)
;; get-evens:
;;   inputs: a list L of numbers
;;   outputs: a new list that contains only the even numbers of L
;;             (in the same order that the appear in L)
(define (get-evens L))

;; tests
(check-equal? (get-evens '()) '())
(check-equal? (get-evens '(1)) '())
(check-equal? (get-evens '(1 2 3)) '(2))
(check-equal? (get-evens '(1 2 3 4)) '(2 4))

;; get-odds:
;;   inputs: a list L of numbers
;;   outputs: a new list that contains only the odd numbers of L
;;             (in the same order that the appear in L)
(define (get-odds L))

;; tests
;;; count-ones:
;;; inputs: a list L of numbers
;;; outputs: the number of times the value 1 appears in L
(define (count-ones L)

; tests
(check-equal? (count-ones '()) 0)
(check-equal? (count-ones '(1)) 1)
(check-equal? (count-ones '(1 2 1)) 2)
(check-equal? (count-ones '(0 0 121 0)) 0)

;;; deep-count-ones:
;;; inputs: a list L of elements
;;; outputs: the number of times the value 1 appears
;;; in L (including in any sub-lists of L)

; tests
(check-equal? (deep-count-ones '()) 0)
(check-equal? (deep-count-ones '(1)) 1)
(check-equal? (deep-count-ones '(1 2 1)) 2)
(check-equal? (deep-count-ones '(0 0 121 0)) 0)
(check-equal? (deep-count-ones '((1))) 1)
(check-equal? (deep-count-ones '((1) 2 ((1)))) 2)
(check-equal? (deep-count-ones '((0 0 121 0))) 0)