```racket
#lang racket

(struct empty-list () #:transparent)

(define empty (empty-list))
empty

(struct cons-list (v prev) #:transparent)

(cons-list 5 empty)

;(cons-list 5 5)

(define (make-list args)
  (match args
    ((list)(empty-list))
    ((list i (? empty-list? x))(cons-list i x))
    ((list i (? cons-list? x))(cons-list i x))
    ((list i)(cons-list i (empty-list)))
    (_ (printf "Error: not a valid list!\n"))))

(make-list (list ))
(make-list (list 1))
(make-list (list 2 empty))
(make-list (list 1 2 3 4))

(define (get-first-list lst)
  (match lst
    ((? empty-list? lst) (printf "Error: not long enough!\n"))
    ((cons-list v rest) v)))

(define alpha (make-list (list "c" (make-list (list "b" (make-list (list "a")))))))
alpha
```

```
(get-first-list alpha)

(define (get-rest-list lst)
  (match lst
    ((? empty-list? lst) (printf "Error: not long enough!\n"))
    ((cons-list v rest) rest)))

(get-rest-list alpha)

(define (get-item-index lst i)
  (letrec ((helper (lambda (lst i j)
                     (if (= i j)
                         (get-first-list lst)
                         (helper (get-rest-list lst) i (+ j 1))))))
    (helper lst i 0)))
(get-item-index alpha 1)

(define (length-list lst)
  (cond ((empty-list? lst) 0)
        ((cons-list? lst) (+ 1
                             (length-list (get-rest-list lst))))
        (else (printf "Error: not a list!"))))

(length-list alpha)

(define (length-list-1 lst)
  (letrec ((helper (lambda (l len)
                     (if (empty-list? l)
                         len
                         (helper (get-rest-list l)
                                 (+ 1 len))))))
    (helper lst 0)))

(length-list-1 alpha)

(define (length-list-2 lst)
  (match lst
    ((empty-list) 0)
    ((cons-list _ rest) (+ 1 (length-list-2 rest)))
    (_ "Error: not a list!")))

(length-list-2 alpha)
```