```racket
;; 1
(define (sequence low high stride)
  (if (> low high)
      null
      (cons low (sequence (+ low stride) high stride))))

;; 1 - use lambda is OK
(define sequence2 (lambda (low high stride)
  (if (> low high)
      null
      (cons low (sequence (+ low stride) high stride)))))

;; 2
(define (string-append-map xs suffix)
  (map (lambda (s) (string-append s suffix)) xs))

;; 3
(define (list-nth-mod xs n)
  (cond  (< n 0) (error "list-nth-mod: n must be non-negative")
         (null? xs) (error "list-nth-mod: list must be non-empty")
         #t (let* ( len (length xs)
                    posn (remainder n len) )
              (car (list-tail xs posn))) ))
;; 4
(define (stream-for-n-steps s n)
  (if (= n 0)
      null
      (let ( next (s) )
         (cons (car next) (stream-for-n-steps (cdr next) (- n 1))))))
;; 5
(define funny-number-stream
  (letrec ( f (lambda (n) (cons (if (= (remainder n 5) 0) (- n) n)
                                (lambda () (f (+ n 1))))) )
    (lambda () (f 1))))
;; 6
(define dan-then-dog
  (letrec ( dan-st (lambda () (cons "dan.jpg" dog-st))
            dog-st (lambda () (cons "dog.jpg" dan-st)) )
    dan-st))

(define (dan-then-dog2)
  (cons "dan.jpg"
        (lambda () (cons "dog.jpg" dan-then-dog2))))

(define dan-then-dog3
  (letrec ( f (lambda (b)
                (if b
                    (cons "dan.jpg" (lambda () (f #f)))
                    (cons "dog.jpg" (lambda () (f #t))))) )
    (lambda () (f #t))))
```

```
;; 7
(define (stream-add-zero s)
  (lambda ()
    (let ( next (s) )
      (cons (cons 0 (car next)) (stream-add-zero (cdr next))))))

;; 8
(define (cycle-lists xs ys)
  (letrec ( loop (lambda (n)
                   (cons (cons (list-nth-mod xs n)
                               (list-nth-mod ys n))
                         (lambda () (loop (+ n 1))))) )
    (lambda () (loop 0))))

;; 9
(define (vector-assoc v vec)
  (letrec ( loop (lambda (i)
                   (if (= i (vector-length vec))
                       #f
                       (let ( x (vector-ref vec i) )
                         (if (and (cons? x) (equal? (car x) v))
                             x
                             (loop (+ i 1)))))) )
    (loop 0)))
;; 10
(define (cached-assoc lst n)
  (let ( cache (make-vector n #f)
         next-to-replace 0 )
    (lambda (v)
      (or (vector-assoc v cache)
          (let ( ans (assoc v lst) )
            (and ans
                 (begin (vector-set! cache next-to-replace ans)
                        (set! next-to-replace
                              (if (= (+ next-to-replace 1) n)
                                  0
                                  (+ next-to-replace 1)))
                        ans)))))))

;; 11
(define-syntax while-less
  (syntax-rules (do)
    ((while-less x do y)
      (let ( z x )
        (letrec ( loop (lambda ()
                         (let ( w y )
                           (if (or (not (number? w)) (>= w z))
                               #t
                               (loop)))) )
          (loop))))))
```