```
(define (interleave list1 list2)
  (cond
    ( (null? list1) list2)
    ( (null? list2) list1)
    ( else
      (cons (car list1) (cons (car list2) (interleave (cdr list1) (cdr list2))) )
    )
  )
)

(define (add-to-all item lists)
  (if (null? lists)
    '()
    (cons (cons item (car lists)) (add-to-all item (cdr lists)))
  )
)

(define (list_subset items start end cur)
  (cond
    ( (null? items) '() )
    ( (equal? cur end) '() )
    ( (>= cur start)
      (cons (car items) (list_subset (cdr items) start end (+ cur 1)) )
    )
    (else
      (list_subset (cdr items) start end (+ cur 1))
    )
  )
)

(define (split items)
  (let*
    ( (halfway (ceiling (/ (length items) 2))) )
    ( cons (list_subset items 0 halfway 0) (list_subset items halfway (length items) 0) )
  )
)

def make_accumulator():
    sum = 0
    def accumulator(value):
        nonlocal sum
        sum += value
        return sum
    return accumulator

def memoize(func):
    previous_results = {}
    def memoized_func(*args):
        if args in previous_results:
            return previous_results[args]
        previous_results[args] = func(*args)
        return previous_results[args]
    return memoized_func


def chain(*funcs):
    if len(funcs) == 0:
        return lambda x: x
    if len(funcs) == 1:
        return lambda x: funcs[0](x)
    return lambda x: funcs[0](chain(*funcs[1:])(x))

def scale(items, factor):
    while True:
        yield next(items) * factor
```

```scheme
(define (list-append list1 list2)
  (if (null? list1)
    list2
    (cons (car list1) (list-append (cdr list1) list2))
  )
)

(define (deep-reverse items)
  (if (null? items)
    '()
    (let*
      (
        (first (car items))
        (rest (cdr items))
      )
      (list-append
        (deep-reverse rest)
        (if (list? first)
          (list (deep-reverse first))
          (list first)
        )
      )
    )
  )
)

(define (contains items el)
  (cond
    ((null? items) #f)
    ((equal? el (car items)) #t)
    (else
      (contains (cdr items) el)
    )
  )
)

(define (remove item lst)
    (if (null? lst)
        lst
        (let
            (
                (first (car lst))
                (rest (remove item (cdr lst)))
            )
            (if (eq? first item)
                rest
                (cons first rest)
            )
        )
    )
)

(define (repeated fn n)
    (if (eq? n 0)
        (lambda (x) x)
        (lambda (x)
            (fn ((repeated fn (- n 1)) x))
        )
    )
)
```