A.A. Stepanov - CS 603 Notes

```
;;; table(k) (0<=k<m) represents the primality
;;; of 2k+3

(define (make-sieve-table m)
  (define (mark tab i step m)
    (cond ((> m i)
           (vector-set! tab i #!false)
           (mark tab (+ i step) step m))))
  (define (scan tab k p s m)
    (cond ((> m s)
           (if (vector-ref tab k) (mark tab s p m))
           (scan tab (+ k 1) (+ p 2) (+ s p p 2) m))
          (else tab)))
  (scan (make-vector m #!true) 0 3 3 m))

;;; 2 <= n <= 20000

(define (sieve n)
  (let ((m (quotient (- n 1) 2)))
    (define (loop tab k p result m)
      (if (<= m k)
          (reverse! result)
          (let ((r (if (vector-ref tab k)
                       (cons p result)
                       result)))
            (loop tab (+ k 1) (+ p 2) r m))))
    (loop (make-sieve-table m) 0 3 (list 2) m)))
```

```
;;; and we can do a generic version of the same


(syntax (bit-set! a b) (vector-set! a b #!false))

(syntax (bit-ref a b) (vector-ref a b))

(syntax (make-bit-table a) (make-vector a #!true))

(define (make-sieve-table m)
  (define (mark tab i step m)
    (cond ((> m i)
           (bit-set! tab i)
           (mark tab (+ i step) step m))))
  (define (scan tab k p s m)
    (cond ((> m s)
           (if (bit-ref tab k) (mark tab s p m))
           (scan tab (+ k 1) (+ p 2) (+ s p p 2) m))
          (else tab)))
  (scan (make-bit-table m) 0 3 3 m))

;;; 2 <= n <= 20000

(define (sieve n)
  (let ((m (quotient (- n 1) 2)))
    (define (loop tab k p result m)
      (if (<= m k)
          (reverse! result)
          (let ((r (if (bit-ref tab k)
                       (cons p result)
                       result)))
            (loop tab (+ k 1) (+ p 2) r m))))
    (loop (make-sieve-table m) 0 3 (list 2) m)))
```

```
(syntax (bit-set! a b) (string-set! a b #\f))

(syntax (bit-ref a b) (char=? (string-ref a b) #\t))

(syntax (make-bit-table a) (make-string a #\t))

(define (make-sieve-table m)
  (define (mark tab i step m)
    (cond ((> m i)
           (bit-set! tab i)
           (mark tab (+ i step) step m))))
  (define (scan tab k p s m)
    (cond ((> m s)
           (if (bit-ref tab k) (mark tab s p m))
           (scan tab (+ k 1) (+ p 2) (+ s p p 2) m))
          (else tab)))
  (scan (make-bit-table m) 0 3 3 m))

(define (sieve n)
  (let ((m (quotient (- n 1) 2)))
    (define (loop tab k p result m)
      (if (<= m k)
          (reverse! result)
          (let ((r (if (bit-ref tab k)
                       (cons p result)
                       result)))
            (loop tab (+ k 1) (+ p 2) r m))))
    (loop (make-sieve-table m) 0 3 (list 2) m)))
```

```
(syntax (bit-set! a b)
        (let ((position (quotient b 8)))
          (let ((byte (char->integer (string-ref a position)))
                (shift (vector-ref '#(1 2 4 8 16 32 64 128)
                                   (modulo b 8))))
            (if (odd? (quotient byte shift))
                (string-set! a position
                             (integer->char (- byte shift)))))))

(syntax (bit-ref a b)
        (let ((byte (char->integer
                     (string-ref a (quotient b 8))))
              (shift (vector-ref '#(1 2 4 8 16 32 64 128)
                                 (modulo b 8))))
          (odd? (quotient byte shift))))

(syntax (make-bit-table a)
        (make-string (ceiling (/ a 8)) (integer->char 255)))

(define (make-sieve-table m)
  (define (mark tab i step m)
    (cond ((> m i)
           (bit-set! tab i)
           (mark tab (+ i step) step m))))
  (define (scan tab k p s m)
    (cond ((> m s)
           (if (bit-ref tab k) (mark tab s p m))
           (scan tab (+ k 1) (+ p 2) (+ s p p 2) m))
          (else tab)))
  (scan (make-bit-table m) 0 3 3 m))

(define (sieve n)
  (let ((m (quotient (- n 1) 2)))
    (define (loop tab k p result m)
      (if (<= m k)
          (reverse! result)
          (let ((r (if (bit-ref tab k)
                       (cons p result)
                       result)))
            (loop tab (+ k 1) (+ p 2) r m))))
    (loop (make-sieve-table m) 0 3 (list 2) m)))



;;;; Pairs

;;; Primitives:

;;; cons: (cons 1 2) ==> (1 . 2)
```

```
;;; car:   (car '(1 . 2)) ==> 1

;;; cdr:   (cdr '(1 . 2)) ==> 2

;;; pair?: (pair? '(1 . 2)) ==> #!true
;;;        (pair? 1)        ==> #!false

;;; set-car!: (define a '(1 . 2)) ==> ??
;;;           (set-car! a 0)      ==> ??
;;;           a                   ==> (0 . 2)
;;;   used to be known as rplaca

;;; set-cdr!: (define a '(1 . 2)) ==> ??
;;;           (set-cdr! a 0)      ==> ??
;;;           a                   ==> (1 . 0)
;;;   used to be known as rplacd

;;;; Lists

;;; Primitives:

;;; Empty list:

;;; ():     '() ==> ()
;;;          (pair? '()) ==> #!false !!! nil is not a pair !!!
;;; used to be known as nil

;;; (1 . (2 . (3 . ()))) ==> (1 2 3)

;;; null?: (null? '()) ==> #!false
;;; used to be known as null

;;; Unlike in LISP (car '()) ==> error
;;;                (cdr '()) ==> error
;;; TI SCHEME does not signal that error, but no code should
depend on
;;; (cdr '()) returning '()

;;; Proper list is a pair cdr of which is either a proper list
;;; or an empty list

;;; Problem:

;;; define a predicate PROPER-LIST?

(define (proper-list? l)
  (if (pair? l)
      (proper-list? (cdr l))
      (null? l)))
```

```
;;; An improper (dotted) list is a chain of pairs not ending in
the empty
;;; list

;;; Problem:

;;; define a predicate IMPROPER-LIST?

(define (last-cdr l)
  (if (pair? l)
      (last-cdr (cdr l))
      l))

(define (improper-list? l)
  (and (pair? l) (not (null? (last-cdr l)))))

;;; More about lambda

;;; there are three ways to specify formal arguments of a function:

;;; 1 - (lambda variable <body>) ==> the procedure takes any
number of
;;;     arguments; they are put in a list and the list is bound
to a
;;;     variable

;;; 2 - (lambda proper-list-of-distinct-variables <body>)
;;;     the procedure takes a fixed number of arguments equal the
length
;;;     of the proper-list-of-distinct-variables; it is an error
to give it
;;;     more or less

;;; 3 - (lambda improper-list-of-distinct-variables <body>)
;;;     the extra arguments are bound to the last variable

;;; Non-primitive (but standard) functions on lists

;;; (define (caar x) (car (car x)))

;;; (define (cadr x) (car (cdr x)))

;;; (define (cdar x) (cdr (car x)))

;;; (define (cddr x) (cdr (cdr x)))

;;; ... and up to four letters

(define list (lambda x x))
```

```
;;; Explain!

;;; Problem:

;;; define a function LENGTH that returns length of a list

(define (my-length l)
  (define (length-loop number list)
    (if (pair? list)
        (length-loop (+ number 1) (cdr list))
        number))
  (length-loop 0 l))

;;; Problem:

;;; define a function REVERSE that returns a newly allocated list
consisting
;;; of the elements of list in reverse order

(define (reverse-append x y)
  (if (pair? x)
      (reverse-append (cdr x) (cons (car x) y))
      y))

(define (my-reverse x)
  (reverse-append x '()))

;;; Equivalence predicates

;;; <see pages 12-14 of R3R>

;;; Destructive functions

;;; reverse returns a new list (a new chain of pairs)
;;; but we may want to reverse the original list

;;; a function F is called applicative iff

;;; (lambda (x) ((lambda (y) (f x) (equal? x y)) (copy x)))

;;; always returns #!true

;;; for an applicative function F a function F! is its destructive
;;; equivalent iff

;;; 1. (f x) == (f! (copy x))

;;; 2. (not (equal? x (f x)))
;;;        implies
```

```
;;;    ((lambda (y) (f x) (not (equal? x y))) (copy x))

;;; from this two axioms we can derive:

;;; Bang rule 1:

;;; (w x) = (f (g x)) => (w! x) = (f! (g! x))

;;; Bang rule 2:

;;; (w! x) = (f! (g! x)) => (w x) = (f! (g x))


;;; Problem:

;;; implement REVERSE!

(define (reverse-append! x y)
  (define (loop a b c)
    (set-cdr! a c)
    (if (pair? b)
        (loop b (cdr b) a)
        a))
  (if (pair? x)
      (loop x (cdr x) y)
      y))

(define (my-reverse! x) (reverse-append! x '()))

;;; it is a little more difficult to right an iterative
;;; procedure COPY-LIST

;;; we can always do

(define (stupid-copy-list l)
  (if (pair? l)
      (cons (car l) (stupid-copy-list (cdr l)))
      l))


;;; as a matter of fact, it is better to define it as:

(define (not-so-stupid-copy-list l)
  (reverse! (reverse l)))

;;; there is a very good way to do it:

(define (rcons x y)
  (set-cdr! x (cons y '()))
  (cdr x))
```

```
(define (copy-list x)
  (define (loop x y)
    (if (pair? y)
        (loop (rcons x (car y)) (cdr y))
        (set-cdr! x y)))
  (if (pair? x)
      ((lambda (header) (loop header (cdr x)) header)
       (list (car x)))
      x))

;;; COPY-LIST is still much slower than NOT-SO-STUPID-COPY-LIST

;;; redefine RCONS as:

(define-integrable
  rcons
  (lambda (x y)
    (set-cdr! x (cons y '()))
    (cdr x)))

;;; and recompile COPY-LIST

;;; Problem:

;;; implement APPEND as a function of an arbitrary number of
lists
;;; which returns a list containing the elements of the first
list
;;; followed by the elements of the other lists
;;; the resulting list is always newly allocated, exept that it
shares
;;; structure with the last list argument. The last argument may
actually
;;; be any object; an improper list results if it is not a proper
list
;;; (see R3R page 16)

(define my-append
  ((lambda (header)
     (lambda lists
       (define (main-loop lists first next last)
         (set-cdr! last first)
         (if next
             (main-loop next
                        (car next)
                        (cdr next)
                        (inner-loop first last))
             (cdr header)))
       (define (inner-loop list last)
```

```
          (if (pair? list)
              (inner-loop (cdr list) (rcons last (car list)))
              last))
        (if lists
            (main-loop lists (car lists) (cdr lists) header)
            '())))
    (list '())))
```

```
;;; Problem:

;;; implement APPEND!

(define my-append!
  ((lambda (header)
     (lambda lists
       (define (main-loop lists first next last)
         (set-cdr! last first)
         (if next
             (main-loop next
                        (car next)
                        (cdr next)
                        (inner-loop first last))
             (cdr header)))
       (define (inner-loop list last)
         (if (pair? list)
             (last-pair list)
             last))
       (if lists
           (main-loop lists (car lists) (cdr lists) header)
           '())))
   (list '())))
```

```
(define (list-copy x)
  (define (loop rest last)
    (cond ((pair? rest)
           (let ((new (list (car rest))))
             (set-cdr! last new)
             (loop (cdr rest) new)))
          (else (set-cdr! last rest))))
  (if (pair? x)
      (let ((first (list (car x))))
        (loop (cdr x) first)
        first)
      x))

(define (vector-copy v)
  (define (loop u n m)
    (cond ((< n m)
           (vector-set! u n (vector-ref v n))
           (loop u (+ n 1) m))
          (else
           u)))
  (let ((l (vector-length v)))
    (loop (make-vector l) 0 l)))

(define (stupid-copy tree)
  (cond ((atom? tree)
         tree)
        (cons (stupid-copy (car tree))
              (stupid-copy (cdr tree)))))

(define (tree-copy tree)
  (define (loop l stack)
    (cond ((pair? (car l))
           (set-car! l (cons (caar l) (cdar l)))
           (loop (car l)
                 (if (pair? (cdr l)) (cons l stack) stack)))
          ((pair? (cdr l))
           (set-cdr! l (cons (cadr l) (cddr l)))
           (loop (cdr l) stack))
          ((pair? stack)
           (let ((i (car stack))
                 (j (cdr stack)))
             (set-car! stack (cadr i))
             (set-cdr! stack (cddr i))
             (set-cdr! i stack)
             (loop stack j)))))
  (if (pair? tree)
      (let ((n (cons (car tree) (cdr tree))))
        (loop n '())
        n)
      tree))
```

```
;;; The problem we are trying to solve is to rotate a vector
;;; to the left by I positions


(define swap!
  (lambda (v i j)
    (let ((temp (vector-ref v i)))
      (vector-set! v i (vector-ref v j))
      (vector-set! v j temp))))

(define subvector-reverse!
  (named-lambda (loop v i j)
    (if (< i j)
        (begin
          (swap! v i j)
          (loop v (+ i 1) (- j 1))))))

(define rotate!
  (lambda (v i)
    (let* ((n (vector-length v))
           (j (modulo i n)))
      (subvector-reverse! v 0 (- j 1))
      (subvector-reverse! v j (- n 1))
      (subvector-reverse! v 0 (- n 1))
      v)))
```

```
(define
  list-length
  length)

(define
  sequence-length
  (lambda (x)
    (cond ((list? x) (list-length x))
          ((vector? x) (vector-length x))
          ((string? x) (string-length x))
          (else (error "Invalid operand to sequence operation"
                       (list 'sequence-length x))))))

(define
  empty?
  (lambda (seq) (zero? (sequence-length seq))))

(define
  sequence-ref
  (lambda (x i)
    (cond ((pair? x) (list-ref x i))
          ((vector? x) (vector-ref x i))
          ((string? x) (string-ref x i))
          (else (error "Invalid operand to sequence operation"
                       (list 'ref x i)))))))

(define
  sequence-set!
  (lambda (x i object)
    (cond ((pair? x) (set-car! (list-tail x i) object))
          ((vector? x) (vector-set! x i object))
          ((string? x) (string-set! x i object))
          (else (error "Invalid operand to sequence operation"
                       (list 'sequence-set! x i object))))))

(define
  make-list
  (lambda (length . object)
    (letrec
      ((loop
        (lambda (length result object)
          (if (<= length 0)
              result
              (loop (- length 1) (cons object result) object)))))
      (loop length '() (if object (car object) '())))))

(define
  sequence-copy
  (lambda (s)
    (cond ((pair? s) (list-copy s))
```

```
          ((vector? s) (vector-copy s))
          ((string? s) (string-copy s))
          (else s))))

(define
  sequence-reverse!
  (lambda (s)
    (letrec
      ((vector-reverse!
         (lambda (v first last)
           (if (>= first last)
               v
               (let ((temp (vector-ref v first)))
                 (vector-set! v first (vector-ref v last))
                 (vector-set! v last temp)
                 (vector-reverse! v (+ first 1) (- last 1))))))
       (string-reverse!
         (lambda (string first last)
           (if (>= first last)
               string
               (let ((temp (string-ref string first)))
                 (string-set!
                   string first (string-ref string last))
                 (string-set! string last temp)
                 (string-reverse!
                   string (+ first 1) (- last 1)))))))
      (cond ((pair? s) (reverse! s))
            ((vector? s)
             (vector-reverse! s 0 (- (vector-length s) 1)))
            ((string? s)
             (string-reverse! s 0 (- (string-length s) 1)))
            (else s)))))

(define
  for-each
  (lambda (operation seq)
    (letrec
      ((list-for-each
         (lambda (operation list)
           (if (pair? list)
               (begin
                 (operation (car list))
                 (list-for-each operation (cdr list))))))
       (vector-for-each
         (lambda (operation v i length)
           (if (< i length)
               (begin
                 (operation (vector-ref v i))
                 (vector-for-each operation v (+ i 1) length)))))
       (string-for-each
```

```
          (lambda (operation string i length)
            (if (< i length)
                (begin
                  (operation (string-ref string i))
                  (string-for-each
                    operation string (+ i 1) length))))))
       (cond ((pair? seq)
              (list-for-each operation seq))
             ((vector? seq)
              (vector-for-each
                operation seq 0 (vector-length seq)))
             ((string? seq)
              (string-for-each
                operation seq 0 (string-length seq)))
             ))))


(define
  map!
  (lambda (operation seq)
    (letrec
      ((list-map!
          (lambda (operation list)
            (if (pair? list)
                (begin
                  (set-car! list (operation (car list)))
                  (list-map! operation (cdr list))))))
        (vector-map!
          (lambda (operation vector i length)
            (if (< i length)
                (begin
                  (vector-set!
                    vector i (operation (vector-ref vector i)))
                  (vector-map!
                    operation vector (+ i 1) length)))))
        (string-map!
          (lambda (operation string i length)
            (if (< i length)
                (begin
                  (string-set!
                    string i (operation (string-ref string i)))
                  (string-map!
                    operation string (+ i 1) length))))))
       (cond ((pair? seq)
              (list-map! operation seq))
             ((vector? seq)
              (vector-map! operation seq 0 (vector-length seq)))
             ((string? seq)
              (string-map! operation seq 0 (string-length seq)))
             )
```

```
      seq)))

(define
  member-if
  (lambda (predicate? list)
    (letrec
      ((loop (lambda (predicate? x)
               (if (pair? x)
                   (if (predicate? (car x))
                       x
                       (loop predicate? (cdr x)))
                   '()))))
      (loop predicate? list))))
```

```
(define
  filter
  (lambda (predicate? list)
    (letrec
      ((loop
         (lambda (predicate? rest result)
           (if (pair? rest)
               (if (predicate? (car rest))
                   (begin
                     (set-cdr! result (cons (car rest) '()))
                     (loop predicate? (cdr rest) (cdr result)))
                   (loop predicate? (cdr rest) result))
               (set-cdr! result rest)))))
      (let ((first (member-if predicate? list)))
        (if (pair? first)
            (let ((result (cons (car first) '())))
              (loop predicate? (cdr first) result)
              result)
            '())))))

(define
  filter!
  (lambda (predicate? list)
    (letrec
      ((loop
         (lambda (predicate? rest next)
           (if (pair? next)
               (if (predicate? (car next))
                   (loop predicate? next (cdr next))
                   (begin
                     (set-cdr! rest (cdr next))
```

```
                      (loop predicate? rest (cdr rest))))))))
      (let ((first (member-if predicate? list)))
        (if (pair? first)
            (begin
              (loop predicate? first (cdr first))
              first)
            '()))))))

(define
  for-each-cdr
  (lambda (operation list)
    (letrec ((loop
               (lambda (rest)
                 (if (pair? rest)
                     (begin (operation rest)
                            (loop (cdr rest)))))))
      (loop list))))

(define
  for-each-cdr!
  (lambda (operation list)
    (letrec ((loop
               (lambda (rest)
                 (if (pair? rest)
                     (let ((temp (cdr rest)))
                       (operation rest)
                       (loop temp))))))
      (loop list))))

(define
  vector-map
  (lambda (operation vector)
    (letrec
      ((loop
         (lambda (operation old new i length)
           (if (< i length)
               (begin
                 (vector-set!
                   new i (operation (vector-ref old i)))
                 (loop operation old new (+ i 1) length))
               new))))
      (let ((length (vector-length vector)))
        (loop operation vector (make-vector length) 0 length)))))

(define
  vector-copy
  (lambda (vector)
    (letrec
      ((loop
         (lambda (old new i length)
```

```
          (if (< i length)
              (begin
                (vector-set! new i (vector-ref old i))
                (loop old new (+ i 1) length))
              new))))
     (let ((length (vector-length vector)))
       (loop vector (make-vector length) 0 length)))))

(define
  map-append!
  (let ((header (list '())))
    (lambda (procedure x)
      (set-cdr! header '())
      (let ((result header))
        (for-each
          (lambda (y)
            (set-cdr! result y)
            (set! result (last-pair result)))
          x)
        (cdr header))))))

(define
  accumulate
  (lambda (operation seq result)
    (for-each
      (lambda (x) (set! result (operation result x)))
      seq)
    result))

(define
  reduce
  (lambda (operation seq)
    (letrec
      ((list-reduce
         (lambda (operation rest result)
           (if (pair? rest)
               (list-reduce
                 operation (cdr rest) (operation
                                        result (car rest)))
               result)))
       (vector-reduce
         (lambda (operation v i length result)
           (if (>= i length)
               result
               (vector-reduce
                 operation
                 v (+ i 1) length (operation
                                    result (vector-ref v i))))))
       (string-reduce
         (lambda (operation s i length result)
```

```
            (if (>= i length)
                result
                (string-reduce
                  operation
                  s (+ i 1) length (operation
                                     result (string-ref v i)))))))
      (cond ((pair? seq)
             (list-reduce operation (cdr seq) (car seq)))
            ((vector? seq)
             (if (not (zero? (vector-length seq)))
                 (vector-reduce
                   operation
                   seq
                   1
                   (vector-length seq)
                   (vector-ref seq 0))
                 '#()))
            ((string? seq)
             (if (not (zero? (string-length seq)))
                 (string-reduce
                   operation
                   seq
                   1
                   (string-length seq)
                   (string-ref seq 0))
                 ""))
            (else '()))))

(define
  right-reduce!
  (lambda (operation seq)
    (reduce operation (sequence-reverse! seq))))

(define
  pairwise-reduce!
  (lambda (operation list)
    (letrec
      ((loop
         (lambda (operation x)
           (if (pair? (cdr x))
               (begin
                 (set-car! x (operation (car x) (cadr x)))
                 (set-cdr! x (cddr x))
                 (loop operation (cdr x)))))))
      (if (pair? list)
          (begin
            (loop operation list)
            list)
          '()))))
```

```
(define
  parallel-reduce!
  (lambda (operation list)
    (letrec
      ((loop
         (lambda (operation x)
           (if (pair? (cdr x))
               (begin
                 (pairwise-reduce! operation x)
                 (loop operation x))
               (car x)))))
      (if (pair? list)
          (loop operation list)
          '())))))

(define (outer-product operation l1 l2)
  (map (lambda (x) (map (lambda (y) (operation x y)) l2)) l1))
```

Tools for sorting study

```
(macro timer
  (lambda (x)
    (let ((exp (cadr x)))
      '(let ((time0 (runtime)))
         ((lambda () ,exp))
         (/ (- (runtime) time0) 100)))))

(define (random-list n . p)
  (if (null? p)
      (let loop ((i 1) (tail '()))
        (if (> i n)
            tail
            (loop (1+ i) (cons (%random) tail))))
      (let loop ((i 1) (tail '()) (p (car p)))
        (if (> i n)
            tail
            (loop (1+ i) (cons (random p) tail) p)))))

(define (random-vector n . p)
  (if (null? p)
      (do ((v (make-vector n))
           (i 0 (+ i 1)))
          ((>= i n) v)
          (vector-set! v i (%random)))
      (do ((p (car p))
           (v (make-vector n))
           (i 0 (+ i 1)))
          ((>= i n) v)
          (vector-set! v i (random p)))))

(define (iota n)
  (let loop ((i (-1+ n)) (tail '()))
    (if (< i 0)
        tail
        (loop (- i 1) (cons i tail)))))

(define (reverse-iota n) (reverse! (iota n)))

(define (random-iota n . p)
  (set! p (if (null? p) n (car p)))
  (let loop ((i (-1+ n)) (tail '()))
    (if (< i 0)
        tail
        (loop (-1+ i) (cons (+ i (random p)) tail)))))

(define (list-copy x) (append x '()))
```

```
(define (make-time-sort copy-function)
  (lambda (sort)
    (gc t)
    (let ((x (copy-function *test-list*)))
      (timer (sort  x >)))))

(define time-sort (make-time-sort list-copy))

(define time-vsort (make-time-sort list->vector))

(define (make-comp-count copy-function)
  (lambda (sort)
    (letrec ((comp-count0 0)
             (comp-count1 0)
             (comp (lambda (x y)
                     (cond ((> 16000 comp-count0)
                            (set! comp-count0 (1+ comp-count0)))
                           (else
                             (set! comp-count1 (1+ comp-count1))
                             (set! comp-count0 1)))
                     (> x y))))
      (sort (copy-function *test-list*) comp)
      (+ comp-count0 (* comp-count1 16000)))))

(define comp-count (make-comp-count list-copy))

(define v-comp-count (make-comp-count list->vector))

(define (make-test x) (set! *test-list* x)
*the-non-printing-object*)

(define *test-list* '())

(define (make-statistic function title-string)
  (lambda (sort length n)
    (do ((nl #\newline)
         (i 0 (1+ i))
         (l '()))
        ((>= i n)
         (for-each
           display
           (list
             "      " title-string nl
             "number of elements: " length nl
             "number of tests: " n nl
             "mean: " (mean l) nl
             "standard-deviation: "  (standard-deviation l) nl))
         *the-non-printing-object*)
```

```
        (make-test (random-list length))
        (set! l (cons (function sort) l)))))


(define statistic-comp-count
  (make-statistic comp-count "COUNTING COMPARISONS"))

(define statistic-v-comp-count
  (make-statistic v-comp-count "COUNTING COMPARISONS"))

(define statistic-time-sort
  (make-statistic time-sort "TIMING"))

(define statistic-time-vsort
  (make-statistic time-vsort "TIMING"))

(define (mean l)
  (let loop ((result 0) (n 0) (l l))
    (if (null? l)
        (/ result n)
        (loop (+ result (car l)) (1+ n) (cdr l)))))

(define (variance l)
  (let ((m (mean l)))
    (let loop ((result 0) (n -1) (l l))
      (if (null? l)
          (/ result n)
          (loop (+ result (let ((i (- (car l) m))) (* i i)))
                (1+ n)
                (cdr l))))))

(define (standard-deviation l) (sqrt (variance l)))

(define (average-deviation l)
  (let ((m (mean l)))
    (let loop ((result 0) (n 0) (l l))
      (if (null? l)
          (/ result n)
          (loop (+ result (abs (- (car l) m))) (1+ n) (cdr
l))))))
```

see Knuth, "The Art of Computer Programming," vol. 3, "Sorting and Searching," pages 105-111.

"the bubble-sort seems to have nothing to recomend it, exept a catchy name" (Knuth)

this is Knuth's version of bubble-sort; it does fewer comparisons than the traditional version, but is more involved; it is much faster than the traditional version for lists that are sorted in the right order, doing only N-1 comparison in such cases.

```
(define (bubble-sort-knuth! v predicate)
  (let loop ((bound (-1+ (vector-length v))))
    (do ((i 0 j)
         (j 1 (1+ j))
         (flag -1))
        ((>= i bound) (if (>= flag 0) (loop flag) v))
        (let ((a (vector-ref v i))
              (b (vector-ref v j)))
          (when (predicate b a)
                (vector-set! v i b)
                (vector-set! v j a)
                (set! flag i)))))))
```

this is the traditional version:

```
(define (bubble-sort! v predicate)
  (do ((n (-1+ (vector-length v)) (-1+ n)))
      ((< n 0) v)
      (do ((i 0 j)
           (j 1 (1+ j)))
          ((>= i n))
          (let ((a (vector-ref v i))
                (b (vector-ref v j)))
            (when (predicate b a)
                  (vector-set! v i b)
                  (vector-set! v j a))))))
```

COMPARISONS COUNTING

comparison counting sort has been known from times immemorial; it was first mentioned by E. H. Field (Journal of ACM, 3, 1956)

this sort does not have any nice properties. It is very slow, and there is no way to improve it. see Knuth, "The Art of Computer Programming," vol. 3, "Sorting and Searching," pages 76-78.

```
(define (comparison-counting-sort v predicate)
  (let ((n (vector-length v)))
    (define counters (make-vector n 0))
    (define sorted (make-vector n))
    (define (bump i)
      (vector-set! counters i (1+ (vector-ref counters i))))
    (do ((i (-1+ n) (-1+ i)))
        ((>= 0 i))
        (do ((j (-1+ i) (-1+ j)))
            ((> 0 j))
            (if (predicate (vector-ref v i) (vector-ref v j))
                (bump j)
                (bump i))))
    ;;sometimes we may just want to output vector COUNTERS
    (do ((i (-1+ n) (-1+ i)))
        ((> 0 i) sorted)
        (vector-set!
          sorted
          (vector-ref counters i)
          (vector-ref v i)))))
```

;DISTRIBUTION COUNTING

this sort is very important for sorting large lists with keys from a small range; it is also used with radix sorting. This particular way to do distribution counting was developed by H. Seward in his MS thesis at MIT in 1954. See Knuth, "The Art of Computer Programming," vol. 3, "Sorting and Searching," pages 76-78.

```
(define (distribution-counting-sort v key-function d)
  ;;v - is a vector, key-function - a function which maps
  ;;elements of this vector into integers x such that 0 =< x < d
  (let* ((n (vector-length v))
         (counter (make-vector d 0))
         (sorted (make-vector n)))
    (do ((j 0 (1+ j)))
        ((>= j n))
        (let ((k (key-function (vector-ref v j))))
          (vector-set! counter k (1+ (vector-ref counter k)))))
     (vector-set! counter 0 (-1+ (vector-ref counter 0)))
    (do ((i 1 (1+ i)))
        ((>= i d))
        (vector-set! counter i
                     (+ (vector-ref counter i)
                        (vector-ref counter (-1+ i)))))
    (do ((j (-1+ n) (-1+ j)))
        ((> 0 j) sorted)
        (let* ((r (vector-ref v j))
               (k (key-function r))
               (i (vector-ref counter k)))
          (vector-set! sorted i r)
          (vector-set! counter k (-1+ i))))))

(define (distribution-by-lists-sort! list key-function d)
  ;;list - is a list, key-function - a function which maps
  ;;elements of this list into integers x such that 0 =< x < d
  (let ((counter (make-vector d 0)))
    (let loop ((i list))
      (if (pair? i)
        (let ((next (cdr i))
              (k (key-function (car i))))
          (set-cdr! i (vector-ref counter k))
          (vector-set! counter k i)
          (loop next))))
    (do ((i (-1+ d) (-1+ i))
         (result '()))
        ((> 0 i) result)
        (let revappend! ((x (vector-ref counter i)))
          (if (pair? x)
              (let ((next (cdr x)))
                (set-cdr! x result)
```

```
                (set! result x)
                (revappend! next)))))))
```

see Knuth, "The Art of Computer Programming," vol. 3, "Sorting and Searching," pages 80-84 and 95-99.

insertion sort (or sift sort) is the best sort to sort sequences which are almost sorted in the right direction other than that it should not be used for sorting sequences with more than 50 elements

```
(define (insertion-vector-sort! v predicate)
  (define last (-1+ (vector-length v)))
  (do ((s last (-1+ s)))
      ((<= s 0) v)
    (do ((i s (1+ i))
         (e (vector-ref v (-1+ s))))
        ((or (> i last)
             (predicate e (vector-ref v i)))
         (vector-set! v (-1+ i) e))
        (vector-set! v (-1+ i) (vector-ref v i)))))

(define (insert! cons list predicate)
  (let ((e (car cons)))
    (cond ((or (null? list)
               (predicate e (car list)))
           (set-cdr! cons list)
           cons)
          (else
            (do ((first list second)
                 (second (cdr list) (cdr second)))
                ((or (null? second)
                     (predicate e (car second)))
                 (set-cdr! first cons)
                 (set-cdr! cons second)
                 list))))))

(define (insertion-list-sort! list predicate)
  (let loop ((l list) (output '()))
    (if (null? l)
        output
        (let ((next (cdr l)))
          (loop next (insert! l output predicate))))))

(define (insertion-sort! x predicate)
  (cond ((pair? x)
         (insertion-list-sort! x predicate))
        ((vector? x)
         (insertion-vector-sort! x predicate))
        (else x)))
```

see

Knuth, "The Art of Computer Programming," vol. 3, "Sorting and Searching," pages 114-123.

this version of quicksort can be used only with non-reflexive test predicates, such as > or <. An attempt to use it with reflexive test predicates, such as >= or <= may result in an out-of-bound vector access.

```
(define (quicksort! v test)
  (define length (vector-length v))
  (let partition ((f 0) (l (-1+ length)))
    (define key
      (let ((a (vector-ref v f))
            (b (vector-ref v l))
            (c (vector-ref v (quotient (+ f l) 2))))
        (cond ((test a b) (cond ((test b c) b)
                                ((test a c) c)
                                (else a)))
              ((test a c) a)
              ((test b c) c)
              (else b))))
    (define (increase i)
      (if (not (test (vector-ref v i) key))
          i
          (increase (1+ i))))
    (define (decrease i)
      (if (not (test key (vector-ref v i)))
          i
          (decrease (-1+ i))))
    (when
      (> (- l f) 8)
      (do ((f-pointer (increase f) (increase (1+ f-pointer)))
           (l-pointer (decrease l) (decrease (-1+ l-pointer))))
          ((>= f-pointer l-pointer)
           (if (= f-pointer l-pointer)
               (if (= f f-pointer)
                   (set! f-pointer (+ f-pointer 1))
                   (set! l-pointer (- l-pointer 1))))
           (cond ((> (- l-pointer f) (- l f-pointer))
                  (partition f-pointer l)
                  (partition f l-pointer))
                 (else
                   (partition f l-pointer)
                   (partition f-pointer l))))
          (let ((temp (vector-ref v f-pointer)))
```

```
            (vector-set! v f-pointer (vector-ref v l-pointer))
            (vector-set! v l-pointer temp)))))
```

the following is just a version of insertion sort which works with non-reflexive tests

```
  (do ((s (- length 2) (- s 1)))
      ((< s 0) v)
      (do ((i s next)
           (next (+ s 1) (+ next 1))
           (e (vector-ref v s)))
          ((or (>= next length)
               (not (test (vector-ref v next) e)))
           (vector-set! v i e))
          (vector-set! v i (vector-ref v next)))))
```

This is a version of quicksort used by MIT Scheme:

```
(define (qsort obj pred)
  (if (vector? obj)
      (qsort! (vector-copy obj) pred)
      (vector->list (qsort! (list->vector obj) pred))))

(define qsort!
  (let ()

    (define (exchange! vec i j)
      (let ((a (vector-ref vec i)))
        (vector-set! vec i (vector-ref vec j))
        (vector-set! vec j a)))

    (named-lambda (qsort! obj pred)
      (define (sort-internal! vec l r)
        (cond ((<= r l) vec)
              ((= r (1+ l))
               (if (pred (vector-ref vec r) (vector-ref vec l))
                   (exchange! vec l r))
              vec)
              (else (quick-merge vec l r))))
      (define  (quick-merge vec l r)
        (let ((first (vector-ref vec l)))
          (define (increase-i i)
            (if (or (> i r) (pred first (vector-ref vec i)))
                i
                (increase-i (1+ i))))
          (define (decrease-j j)
            (if (or (<= j l)
                    (not (pred first (vector-ref vec j))))
                j
                (decrease-j (-1+ j))))
          (define (loop i j)
            (if (< i j)
                (begin (exchange! vec i j)
                       (loop (increase-i (1+ i))
                             (decrease-j (-1+ j))))
                (begin
                  (cond ((> j l)
                         (exchange! vec j l)))
                  (sort-internal! vec (1+ j) r)
                  (sort-internal! vec l (-1+ j)))))
          (loop (increase-i (1+ l))
                (decrease-j r))))
      (if (vector? obj)
          (begin (sort-internal! obj 0 (-1+ (vector-length obj)))
                 obj)
```

```
          (error "QSORT! works on vectors only: " obj)))))

(define (treesort unsorted predicate . k)
  (define n (vector-length unsorted))
  (define sorted '())
  (define m (make-vector (* n 2)))
  (define (minimum i)
    (vector-set!
      m
      i
      (let* ((j (* i 2))
             (first (vector-ref m j))
             (second (vector-ref m (1+ j))))
        (cond ((null? first) second)
              ((null? second) first)
              ((predicate (car first) (car second))
               first)
              (else
                second)))))
  (set! k (if (or (null? k) (> (car k) n)) n (car k)))
  (set! sorted (make-vector k))
  (do ((i 1 (1+ i))
       (tag n (1+ tag)))
      ((> i n))
      (vector-set! m tag (cons (vector-ref unsorted (-1+ i))
                               tag)))
  (do ((i (-1+ n) (-1+ i)))
      ((>= 0 i))
      (minimum i))
  (do ((j 0 (1+ j))
       (i (cdr (vector-ref m 1)) (cdr (vector-ref m 1))))
      ((>= j k) sorted)
      (vector-set! sorted j (car (vector-ref m 1)))
      (vector-set! m i '())
      (do ((i (quotient i 2) (quotient i 2)))
          ((<= i 0))
          (minimum i))))
```

We shall first consider merge-sort. This will lead us to several new functional forms and allow us at first to produce a more efficient code for merge-sort itself and then to produce a new sorting algorithm which has some very unusual properties.

Recursive Merge-Sort.

The traditional version of merge-sort is based on the divide-and-conquer programming paradigm. First, we split the list of items in two halves, merge-sort them separately, and then merge them together. The following is the SCHEME translation of a COMMON LISP code from Winston and Horn:

```
(define (winston-sort x predicate)
  (define (merge a b)
    (cond ((null? a) b)
          ((null? b) a)
          ((predicate (car a) (car b))
           (cons (car a) (merge (cdr a) b)))
          (else
            (cons (car b) (merge a (cdr b))))))
  (define (head l n)
    (cond ((negative? n) '())
          (else (cons (car l) (head (cdr l) (- n 2))))))
  (define (tail l n)
    (cond ((negative? n) l)
          (else (tail (cdr l) (- n 2)))))
  (define (first-half l) (head l (- (length l) 1)))
  (define (last-half l) (tail l (- (length l) 1)))
  (cond ((null? (cdr x)) x)
        (else (merge (winston-sort (first-half x) predicate)
                     (winston-sort (last-half x) predicate)))))
```

Splitting linked lists in two is a time consuming activity. The same list is traversed twice at first by FIRST-HALF and then by SECOND-HALF, not counting two traversals by LENGTH.

Improving merge.

The traditional merge algorithm can be implemented thus:

```
(define (merge! l1 l2 predicate)
  (define (merge-loop l1 l2 last)
    (cond ((null? l1) (set-cdr! last l2))
          ((null? l2) (set-cdr! last l1))
          ((predicate (car l1) (car l2)) (set-cdr! last l1)
           (merge-loop (cdr l1) l2 l1))
          (else (set-cdr! last l2)
                (merge-loop l1 (cdr l2) l2))))
  (cond ((null? l1) l2) ;we do not need NULL tests for sorting
        ((null? l2) l1)
        ((predicate (car l1) (car l2))
         (merge-loop (cdr l1) l2 l1) l1)
        (else (merge-loop l1 (cdr l2) l2) l2)))
```

It can be seen that one of NULL? tests in MERGE-LOOP is unneeded. Only the list which was advanced during previous iteration can be empty. And we can keep this information around by putting the one which advanced as a first argument to the tail-recursive process which does the merging. That immediately allows us to reduce the number of pointer manipulations by a factor of two, since we need to do SET-CDR! only when the previous winner loses. All that allows us to come up with:

```
(define (unstable-merge! l1 l2 predicate)
  (define (merge-loop i j)
    (let ((k (cdr i)))
      (cond ((null? k) (set-cdr! i j))
            ((predicate (car k) (car j)) (merge-loop k j))
            (else (set-cdr! i j) (merge-loop j k)))))
  (cond ((null? l1) l2)
        ((null? l2) l1)
        ((predicate (car l1) (car l2))
         (merge-loop l1 l2) l1)
        (else (merge-loop l2 l1) l2)))
```

we can make this merge stable by using alterating loops:

```
(define merge!
  (lambda (l1 l2 predicate)
    (letrec
      ((right-loop
         (lambda (i j)
           (let ((k (cdr i)))
             (cond ((null? k) (set-cdr! i j))
                   ((predicate (car k) (car j)) (right-loop k j))
                   (else (set-cdr! i j) (wrong-loop j k))))))
       (wrong-loop
         (lambda (i j)
           (let ((k (cdr i)))
             (cond ((null? k) (set-cdr! i j))
                   ((predicate (car j) (car k))
                    (set-cdr! i j) (right-loop j k))
                   (else (wrong-loop k j)))))))
      (cond ((null? l1) l2)
            ((null? l2) l1)
            ((predicate (car l1) (car l2))
             (right-loop l1 l2) l1)
            (else (wrong-loop l2 l1) l2)))))


(define
  merge
  (lambda (x y predicate)
    (letrec
      ((loop
         (lambda (first second result)
           (cond ((null? first)
                  (reverse! (reverse-append second result)))
                 ((null? second)
                  (reverse! (reverse-append first result)))
                 ((predicate (car first) (car second))
                  (loop (cdr first) second (cons (car first)
                                                 result)))
                 (else
                   (loop first (cdr second) (cons (car second)
                                                  result)))))))
      (loop x y '()))))
```

It can be easily seen that we can sort a list by first transforming it into a list of one element lists and then reducing merge on it:

```
(define (?-sort! l predicate)
  (reduce (lambda (x y) (merge! x y predicate)) (listify! l)))
```

where LISTIFY! is:

```
(define (listify! l) (map! list l))
```

And our ?-sort! sorts. But it sorts extremely slowly. This sequence of merges transforms merge-sort into insertion-sort.

It is now easy to see that what we need is another reduction operator. Instead of reducing the list from left to right (or from right to left - both orders are possible in COMMON LISP) we want to reduce the list in a tournament fashion - with logN rounds. We can do it with the help of the following two functional forms:

```
(define (pairwise-reduce! operation l)
  (let loop ((x l))
    (cond ((null? (cdr x)) l)
          (else (set-car! x (operation (car x) (cadr x)))
                (set-cdr! x (cddr x)) (loop (cdr x))))))

(define (parallel-reduce! operation l)
  (if (null? (cdr l)) (car l)
      (parallel-reduce! operation
                        (pairwise-reduce! operation l))))
```

PARALLEL-REDUCE! is an iterative analog of divide-and-conquer. When used with an associative operation, such as merge, it produces the same result as REDUCE, but very often more quickly. For non-associative operations it produces a different result, which may be valuable in itself and leads to new algorithms.

Now we can easily implement merge-sort:

```
(define (merge-sort! l predicate)
  (parallel-reduce! (lambda (x y) (merge! x y predicate))
           (listify! l)))
```

It can be seen that all the processes involved are iterative and all function calls can be easily removed. We generate exactly N extra conses. But the number of extra conses can be further reduced if LISTIFY! will make not a list of one element lists, but a list of sorted lists with 8 elements each created with the help of the insertion sort. While this can be done, this does not really improve the performance since LISTIFY! takes a very small percentage of total time declining when N grows.

```
(define (put-in-adder! x register function zero)
  (let ((y (car register)) (z (cdr register)))
    (cond ((eqv? y zero) (set-car! register x))
          (else (set-car! register zero)
                (set! x (function x y))
                (if (null? z) (set-cdr! register (list x))
                    (put-in-adder! x z function zero))))))
```

It can be used for many different things from simulating binary 1+ to implementing binomial queues.

We can now define a new version of merge-sort:

```
(define (adder-merge-sort! l predicate)
  (define register (list '()))
  (define (local-merge! x y) (merge! y x predicate))
  (define (local-put-in-adder! x)
    (set-cdr! x '())
    (put-in-adder! x register local-merge! '()))
  (for-each-cdr! local-put-in-adder! l)
  (reduce local-merge! register))
```

It generates logN conses, and is very quick.

```
(define (v-put-in-adder! x register function zero)
  ;;we assume that register is long and there will be no overflow
  (let loop ((x x) (i 0))
    (let ((y (vector-ref register i)))
      (cond ((eqv? y zero) (vector-set! register i x))
            (else (vector-set! register i zero)
                  (loop (function x y) (1+ i)))))))

(define v-adder-merge-sort!
  (let ((register (make-vector 32)))
    (lambda (l predicate)
      (define function (lambda (x y) (merge! y x predicate)))
      (vector-fill! register '())
      (for-each-cdr!
        (lambda (x)
          (set-cdr! x '())
          (v-put-in-adder! x register function '()))
        l)
      (vector-reduce function register))))
```

This is a very fast hand optimized version of mergesort:

```
(define (merge-sort! x predicate)
  (define (merge i j)
    (let ((k (cdr i)))
      (cond ((null? k) (set-cdr! i j))
            ((predicate (car k) (car j)) (merge k j))
            (else (set-cdr! i j) (merge j k)))))
  (do ((l x (cdr l)))
      ((null? l))
      (set-car! l (list (car l))))
  (do ()
      ((null? (cdr x)) (car x))
      (do ((l x (cdr l)))
          ((null? (cdr l)))
          (let ((i (car l))
                (j (cadr l)))
            (cond ((predicate (car i) (car j)) (merge i j))
                  (else (set-car! l j) (merge j i))))
          (set-cdr! l (cddr l)))))
```

This is a Scheme version of MIT MACLISP sort:

```
(define (maclisp-sort! x predicate)

  (define header (list 'hunoz))

  (define (prefix height)
    (cond ((null? x) '())
          ((<? height 1)
           (let ((i x) (j (cdr x)))
             (set-cdr! x '())
             (set! x j)
             i))
          (else
            (merge (prefix (- height 1))
                   (prefix (- height 1))))))

  (define (merge l1 l2)
    (let ((p header))
      (let loop ()
        (cond ((null? l1) (set-cdr! p l2) (cdr header))
              ((null? l2) (set-cdr! p l1) (cdr header))
              ((predicate (car l2) (car l1))
               (set-cdr! p l2)
               (set! p l2)
               (set! l2 (cdr l2))
               (loop))
              (else
                (set-cdr! p l1)
                (set! p l1)
                (set! l1 (cdr l1))
                (loop))))))

  (do ((height -1 (+ height 1))
       (sofar '() (merge sofar (prefix height))))
      ((null? x) sofar)))
```

```
(define (grab x y)
  (set-cdr! x (cons y (cdr x)))
  x)

(define (make-tournament-play predicate)
  (lambda (x y)
    (if (predicate (car x) (car y))
        (grab x y)
        (grab y x))))

(define (make-tournament reduction)
  (lambda (forest predicate)
    (reduction
      (make-tournament-play predicate)
      forest)))

(define sequential-tournament! (make-tournament right-reduce!))

(define parallel-tournament! (make-tournament parallel-reduce!))

(define (make-tournament-sort! tournament1 tournament2)
  (lambda (plist predicate)
    (let ((p (tournament1 (map! list plist) predicate)))
      (for-each-cdr
        (lambda (x) (set-cdr! x (tournament2 (cdr x) predicate)))
        p)
      p)))

(define tournament-sort-p!
  (make-tournament-sort! parallel-tournament!
     parallel-tournament!))

(define tournament-sort-s!
  (make-tournament-sort! parallel-tournament!
     sequential-tournament!))

(define tournament-sort-s-s!
  (make-tournament-sort! sequential-tournament!
       sequential-tournament!))
```

```
(macro grab!
  (lambda (body)
    (let ((x (cadr body))
          (y (caddr body))
          (z (gensym))
          (w (gensym)))
      '(let ((,z ,x) (,w ,y))
         (set-cdr! ,w (cdar ,z))
         (set-cdr! (car ,z) ,w)
         ,z))))

(macro tournament-play!
  (lambda (body)
    (let ((x (cadr body))
          (y (caddr body))
          (predicate (cadddr body)))
      '(if (,predicate (caar ,x) (caar ,y))
           (grab! ,x ,y)
           (grab! ,y ,x)))))

(define (sequential-tournament! forest predicate)
  (cond
    ((null? forest) '())
    ((null? (cdr forest)) (car forest))
    (else
      (let ((x (reverse! forest)))
        (do ((result x (tournament-play! result next predicate))
             (next (cdr x) after-next)
             (after-next (cddr x) (cdr after-next)))
            ((null? after-next)
             (car (tournament-play! result next predicate))))))))

(define (parallel-tournament! forest predicate)
  (define (tournament-round! so-far to-be-done)
    (cond ((null? to-be-done) so-far)
          ((null? (cdr to-be-done))
           (set-cdr! to-be-done so-far)
           to-be-done)
          (else
            (let* ((i (cdr to-be-done))
                   (j (cdr i))
                   (new (tournament-play! to-be-done
                                          i
                                          predicate)))
              (set-cdr! new so-far)
              (tournament-round! new j)))))
  (if (null? forest)
      '()
      (do ((x forest (tournament-round! '() x)))
          ((null? (cdr x)) (car x)))))
```

VECTOR UTILITIES

(vector-last v) - returns the index of the last element in a vector.

(vector-swap! v i j) - interchanges the values of elements i and j in a vector.

(vector-reverse! v) - reverses a vector in place (destructively).

(vector-move! v to from) - move the value from element from to element to.

(vector-compare predicate v first second) - compare element first with element second using predicate.

```
(define-integrable (vector-last v)
  (-1+ (vector-length v)))

(define-integrable (vector-swap! v i j)
  (let ((temp (vector-ref v i)))
    (vector-set! v i (vector-ref v j))
    (vector-set! v j temp)))

(define (vector-reverse! v)
  (do ((first 0 (1+ first))
       (last (vector-last v) (-1+ last)))
      ((>= first last) v)
      (vector-swap! v first last)))

(define-integrable (vector-move! v to from)
  (vector-set! v to (vector-ref v from)))

(define-integrable (vector-compare predicate v first second)
  (predicate (vector-ref v first) (vector-ref v second)))
```

SIFTING

Sift is an algorithmic primitive which can be used to build a variety of sorting algorithms. It is a generalization of the bubbling operation in heaps. Given a vector, v, containing elements to be sorted, sift considers chains of elements. A chain is a sequence of elements whose indices in the vector are related functionally to one another. When bubbling up in an ordinary heap, for example, the next element in a chain has an index which is found by halving the current index. Sift also takes a value whose proper place within the chain is to be found. The proper place of a value within a chain is defined by a predicate, which is used to compare pairs of values. If (predicate a b) is satisfied, then a belongs ahead of b in the chain. Usually, the value passed to sift is a value already in the chain and currently out of place with respect to the predicate. Sift is invoked with this value and with a chain which is otherwise correct with respect to the predicate. After sifting, this value is in the correct place in the chain. Thus, a proper chain with one more element has been created. Starting with chains containing one element (which are trivially correct), sift is called to create larger chains which lead to a variety of structures useful in sorting. Examples of these are heaps (of many kinds), and partially sorted subsequences of elements. As we will see below, many variants of heapsort, shellsort, and selection sort can be created using sift.

(sift v position next-function value fill-pointer predicate) -
- v - vector containing values to be sorted.
- current - position in v where sift is to start.
- next-function - function which returns the position of the next element to be considered in the sift; returns null if current position is the last element to be considered.
- value - the value to be placed in v.
- fill-pointer - last occupied position in v.
- predicate - predicate indicating ordering desired by the sort; i.e., (predicate v[i] v[j]) is satisfied for i < j at the end of the sort.

(sift-all! v step-function start fill-pointer predicate) -
iteratively invokes sift starting from positions start,start-1,... 0. This can be used to set up a heap, do an insertion sort, or do one phase of Shellsort.

```
(define (sift! v current next-function value fill-pointer
            predicate)
 (let ((next (next-function v current fill-pointer predicate)))
  (cond ((or (null? next) (predicate value (vector-ref v next)))
      (vector-set! v current value))
     (else (vector-set! v current (vector-ref v next))
        (sift! v next next-function value fill-pointer
    predicate)))))

(define (sift-all! v next-function start fill-pointer predicate)
 (do ((i start (- i 1)))
   ((< i 0) v)
   (sift! v i next-function (vector-ref v i) fill-pointer
  predicate)))
```

INSERTION SORT

To implement Insertion Sort using the sift primitive, we need only define an appropriate next-function.

(insertion-next step) - next-function for insertion sort. Also, suitable for implementing one phase of Shellsort. Generates next postion by adding a constant to current position.

(insertion-step-sort! v step predicate) - uses insertion-next and sift-all! to sort, or in the case of Sheelsort, to do one phase of a sort by sorting every step-th element in v.

(insertion-sort! v predicate) - Insertion Sort. Invokes insertion-step-sort! with step=1.

```
(define (insertion-step step)
 (lambda (v current fill-pointer predicate)
  (let ((next (+ current step)))
   (if (> next fill-pointer) '() next))))

(define (insertion-step-sort! v step predicate)
 (let ((l (vector-last v)))
  (sift-all! v (insertion-step step) (- l step) l predicate)))

(define (insertion-sort! v predicate)
 (insertion-step-sort! v 1 predicate))
```

SHELLSORT

Refs: D.E. Knuth, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," pp. 84-95.
Donald L. Shell, CACM, Vol. 2, 1959, pp.30-32.
Collected Algorithms from CACM: Algorithm #201

Properties: Sorts vectors in place, not stable, partial sorting not possible, worst case complexity O[N^2], average case complexity varies and is in practice competitve with the best sorts.

Shellsort takes as input a vector of values to be sorted and a sequence of increments. These increments control the sorting process. Each increment is used in turn to define the distance between elements in the vector. Elements in the vector at this distance are considered as a chain (see the description of the sifting operation above) and are sorted. The final increment in the sequence is 1 and so at the end of Shellsort, the vector is totally sorted. Thus, Shellsort can be thought of as a series of insertion sorts. The purpose of the initial sorts in the sequence is to quickly bring elements to positions which are close to the proper positions for these elements so that each individual pass of the algorithm does not have to work too hard it is well known that insertion sort is very fast when the elements to be sorted do not have to move far. Picking a good sequence of increments is an art. We offer several good choices below.

```
(define (make-shellsort! increment-function)
  (lambda (v predicate)
    (for-each
      (lambda (step) (insertion-step-sort! v step predicate))
      (increment-function (vector-length v)))
  v))
```

INCREMENT SEQUENCES FOR SHELLSORT

The following are sequences shown to be good for Shellsort. (Reference: "Handbook of Algorithms and Data Structures", G. H. Gonnet Addison-Wesley, 1984)

(knuth-increments n) - function yielding the sequence recommended

by Knuth in his book. n is the number of elements in the vector of elements to be sorted. The sequence generated is (...., 40, 13, 4, 1). The sequence is generated starting with the value 1 at the end of the sequence. The next (i.e., preceding) value is generated from the current one by multiplying by 3 and adding 1. The final (first) element in the sequence is the largest such number which is less than n.

(shellsort-knuth! v predicate) - Shellsort using Knuth increments.

(pratt-increments n) - increments by shown by Pratt to guarantee O[n * (log (n)^2)] worst case preformance but very slow in practice. Elements of the sequence are composites of powers of 2 and powers of 3. For example if n is 50, the sequence is (48,36,32,27,24,18,16,12,9,6,4,3,2,1).

(shellsort-pratt! v predicate) - Shellsort using Pratt increments.

(gonnet-increments n) - increments recommended by Gonnet in his book. The sequence is generated by starting with floor(.4545n) and continuing to take floor(.4545i) until 1 is reached.

(shellsort-gonnet! v predicate) - Shellsort using Gonnet increments.

(stepanov-increments n) - increments recommended by A. Stepanov. The sequence is generated by taking floor(e^i + .5); i.e., powers of e rounded to the nearest integer. Again, the sequence is generated in reverse order and ends with the largest such value less than n. These increments are the most efficient ones we have found thus far.

(shellsort-stepanov! v predicate) - Shellsort using Stepanov increments.

```
(define (knuth-increments n)
  (do ((i 1 (+ (* i 3) 1))
       (tail '() (cons i tail)))
      ((>= i n) (or (cdr tail) tail))))

(define shellsort-knuth! (make-shellsort! knuth-increments))

(define (pratt-increments n)
  (define (powers base n)
    (do ((x 1 (* x base))
         (result '() (cons x result)))
        ((>= x n) result)))
  (filter (lambda (x) (< x n))
          (parallel-reduce!
            (lambda (x y) (merge! x y >))
            (outer-product * (powers 2 n) (powers 3 n)))))

(define shellsort-pratt! (make-shellsort! pratt-increments))
```

```
(define (gonnet-increments n)
  (define (gonnet n) (floor (* n .45454)))
  (do ((i (gonnet n) (gonnet i))
       (result '() (cons i result)))
      ((>= 1 i) (reverse! (cons 1 result)))))

(define shellsort-gonnet! (make-shellsort! gonnet-increments))

(define (stepanov-increments n)
  (do ((i 1 (+ i 1))
       (e 1 (floor (+ 0.5 (exp i))))
       (tail '() (cons e tail)))
      ((>= e n) tail)))

(define shellsort-stepanov!
  (make-shellsort! stepanov-increments))
```

HEAPS USING SIFTING

Heaps can also be implemented using the sift primitive, inclusing

an entire family of Heapsort algorithms. These algorithms also use some of the vector utilities described above. All of the heap utilities implemented above are reimplemented here using the same names for the functions. Thus, if this entire file is loaded and compiled, these are the functions which will be used, since they the last (most recent) ones defined.

next-functions for sift:

(heap-son v father fill-pointer predicate)

- This is a next-function for sift. Given father, a position in the vector (v, fill-pointer, and predicate are as above in the description of sift) it returns the position of the "larger" successor of father. Thus, if father = i, it returns the false value if 2i+2 is greater than n. (Recall that our vectors are indexed starting from 0; thus a vector of n elements has elements with indices 0,1,...n-1 and the children of an element with index i are those with indices 2i+1 and 2i+2.) It returns 2i+1 if (predicate v[2i+1] v[2i+2]) is true or if 2i+3 is greater than n; and it returns 2i+2 if (predicate v[2i+1] v[2i+2]) is false. This is the appropriate next-function for bubbling down in ordinary heaps.

(heap-up-pointer son) - floor((son-1)/2)

(heap-father v son fill-pointer predicate) - The appropriate next-function for bubbling up in an ordinary heap. It returns (heap-up-pointer son) if son is positive and the false value otherwise.

```
(define (heap-son v father fill-pointer predicate)
  (let ((son (* 2 (1+ father))))
    (cond ((>= fill-pointer son)
           (if (predicate (vector-ref v son)
                          (vector-ref v (-1+ son)))
               son
               (-1+ son)))
          ((= fill-pointer (-1+ son)) (-1+ son))
          (else '()))))

(define (heap-up-pointer son) (quotient (-1+ son) 2))

(define (heap-father v son fill-pointer predicate)
  (if (>= 0 son) '() (heap-up-pointer son)))
```

```
(define (downheap! v father value fill-pointer predicate)
  (sift! v father heap-son value fill-pointer predicate))

(define (upheap! v son value predicate)
  (sift! v son heap-father value son
         (lambda (x y) (predicate y x))))

(define (build-heap! v fill-pointer predicate)
  (sift-all! v heap-son (heap-up-pointer fill-pointer)
  fill-pointer predicate))

(define (heap-set! v position value fill-pointer predicate)
  (if (predicate (vector-ref v position) value)
      (downheap! v position value fill-pointer predicate)
      (upheap! v position value predicate)))
```

HEAPSORT

Williams' Heapsort Algorithm
Refs: Knuth Volume 3 , p. 145-149
Collected Algorithms from CACM: Algorithm #232
CACM, Vol. 7 (1964) pp. 347-348
Properties: sorts vectors in place, not stable, partial sort possible, worst case running time O[N*log(N)].

Heapsort works by setting up a heap. A heap is a binary tree with the following properties. The descendents of node i are nodes 2i and 2i+1. Thus, the links pointing to the descendents of a node are implicit in the nodes' positions in the vector. A node satisfies the predicate (passed as an argument to heapsort) with respect to all its descendents. Thus, for example, if the predicate is <, each node is less than all its descendents. Heapsort begins by building a heap (using build-heap). The heap is built by checking that the predicate is satisfied and interchanging a node with its smaller (in the sense of the predicate) descendent if necessary, so that after the exchange the predicate is satisfied. Traditionally, for the sake of efficiency, the heap is built upside down, in reverse order of the predicate. Here, for clarity, the heap is built right side up. The function of "bubbling down an element, in some cases several levels in the heap, until the predicate is satisfied or the element reaches the bottom of the heap, is handled by downheap. After the heap is set up, the element which should be in the first position in the sorted vector is at the top of the heap (in position 1). The first and last element in the heap are interchanged and the last element is removed from further consideration by decreasing the size of the heap. The new top heap element (taken from the bottom of the heap in the above exchange) is bubbled down. The process of exchange and bubbling is repeated until the entire vector is sorted. At this point, the vector in in reverse order, so reverse! is called to put the vector in the desired sorted order.

(heapsort! v predicate) - Heapsort. v is the vector to be sorted using the predicate.

(read-heap! v fill-pointer predicate) - pop all the elements out of the heap in order.

HEAPSORT USING SIFTING

(heapsort! v predicate) - Heapsort. See description above. This is the traditional version of Heapsort. The heap is built in reverse order of the predicate, which allows the read operation to pop out the elements in reverse order and then place them in their proper positions in the sorted vector when the popped element and the last element in the heap are interchanged.

(read-heap! v fill-pointer predicate) - pop all the elements out of a heap. See description above.

(reverse-heapsort! v predicate) - This is the more natural version of Heapsort, as described in the section above. The heap is built in the natural order and the sorted list is reversed at the end of the sort.

(top-down-build-heap! v fill-pointer predicate) - The heap can be built from the top down. This is useful if the elements are not all available at the time the heap is originally being formed. This has worst case complexity O[nlog(n)].

(top-down-heapsort! v predicate) - Heapsort using top-down-build-heap.

```
(define (read-heap! v fill-pointer predicate)
  (do ((position fill-pointer (-1+ position)))
      ((>= 0 position) v)
      (vector-swap! v position 0)
      (downheap! v 0 (vector-ref v 0) (-1+ position) predicate)))

(define (heapsort! v predicate)
  (build-heap! v (vector-last v) (lambda (x y) (predicate y x)))
  (read-heap! v (vector-last v) (lambda (x y) (predicate y x))))

(define (reverse-heapsort! v predicate)
  (build-heap! v (vector-last v) predicate)
  (read-heap! v (vector-last v) predicate)
  (vector-reverse! v))
```

TOP-DOWN-BUILD-HEAP Top-down-build-heap! allows us to build a heap one element at a time. It is O[N*log(N)] in the worst case and O[N] on the average. We can also implement heapsort with top-down-build-heap!

```
(define (top-down-build-heap! v fill-pointer predicate)
  (do ((position 1 (1+ position)))
      ((> position fill-pointer) v)
      (upheap! v position (vector-ref v position) predicate)))

(define (top-down-heapsort! v predicate)
  (top-down-build-heap! v (vector-last v) predicate)
  (read-heap! v (vector-last v) predicate)
  (vector-reverse! v))
```

3-HEAPS 3-heaps are slightly faster (3% fewer comparisons and 2% less time) than ordinary heaps (2-heaps). In 3-heaps, each non-terminal node has up to 3 children. This results in a shallower tree but requires an additional comparison per level. Of all the possible breadths of heaps, we found 3-heaps to be the best. Note that this section redefines the functions heap-son and heap-up-pointer and should not be loaded unless you intend to use 3-heaps instead of ordinary heaps.

```
(define (heap-son v father fill-pointer predicate)
  (define (test i j)
    (predicate (vector-ref v i) (vector-ref v j)))
  (let ((son (* 3 (1+ father))))
    (cond ((>= fill-pointer son)
           (if (test son (- son 1))
               (if (test son (- son 2)) son (- son 2))
               (if (test (- son 1) (- son 2))
                   (- son 1)
                   (- son 2))))
          ((= fill-pointer (-1+ son))
           (if (test (- son 1) (- son 2)) (- son 1) (- son 2)))
          ((= fill-pointer (- son 2)) (- son 2))
          (else '()))))

(define (heap-up-pointer son) (quotient (-1+ son) 3))
```

D-HEAPS

Using sifting, d-heaps (heaps with d successors per node) can be implemented. This is useful in order to carry out experiments on the relative efficiency of different values of d, which is interesting in the case where there are additions, deletions and changes in value of the vector elements. It is possible, by giving some nodes d children and other d+1 children to form d-heaps for non-integer values of d. We do not do this here, however.

(largest-in-the-range v first last predicate) - returns the largest element between position first and position last, where v[i] is largest if (predicate v[i] v[j]) is true for all j in the range.

(make-d-heap-son d) - returns a heap-son function for a d-heap. For example (define heap-son (make-d-heap-son 4)) sets up the heap-son function for a 4-heap.
(make-d-heap-up-pointer d) - returns a heap-up-pointer function for a d-heap.

```
(define (largest-in-the-range v first last predicate)
  (if (> first last) '()
      (do ((next (1+ first) (1+ next)))
          ((> next last) first)
          (if (predicate (vector-ref v next)
                         (vector-ref v first))
              (set! first next)))))

(define (make-d-heap-son d)
  (lambda (v father fill-pointer predicate)
    (let ((x (* d father)))
      (largest-in-the-range
        v (+ x 1) (min (+ x d) fill-pointer) predicate))))

(define (make-d-heap-up-pointer d)
  (lambda (son) (quotient (-1+ son) d)))

(define (selection-sort! v predicate)
  (do ((last (vector-last v))
       (i 0 (1+ i)))
      ((>= i last) v)
      (vector-swap! v i
                    (largest-in-the-range v i last predicate))))
```

Synactic extensions

So far the only special forms that we used are LAMBDA, IF, DEFINE, QUOTE and SET!

While these forms are powerful enough SCHEME includes several secondary special forms that are normally expressed with the help of the primitive ones.

While SCHEME does not specify a standard mechanism for syntactic expansions actual implementations provide macro mechanism to do the stuff.

Quasiquotation

<see R3R pages 10-11>

Macros

Macro is a function of one argument (macroexpander) associated with a keyword.

When SCHEME compiles an S-expression car of which is a macro keyword it replaces it with a value that is returned by the corresponding macroexpander applied to this S-expression

```
(macro m-square
  (lambda (body)
    '(* ,(cadr body) ,(cadr body))))
```

So if we say

```
(m-square 4)
```

it will expand into

```
(* 4 4).
```

But if we say

```
(m-square (sin 1.234))
```

it will expand into

```
(* (sin 1.234) (sin 1.234))
```

and we are going to evaluate (sin 1.234) twice

```
(macro better-m-square
  (lambda (body)
    (if (or (number? (cadr body))
            (symbol? (cadr body)))
        '(* ,(cadr body) ,(cadr body))
        '((lambda (temp) (* temp temp))
```

```
        ,(cadr body)))))
```

Derived special forms

the simpliest special form we can implement is BEGIN

```
(define (begin-expander body)
  '((lambda () . ,(cdr body))))

(macro my-begin begin-expander)
```

one of the most useful ones is COND

```
(define (cond-expander body)
  (define temp (gensym))
  (define (loop clauses)
    (if (pair? clauses)
        (if (pair? (car clauses))
            (if (eq? 'else (caar clauses))
                '(begin . ,(cdar clauses))
                (if (null? (cdar clauses))
                    '((lambda (,temp)
                        (if ,temp ,temp ,(loop (cdr clauses))))
                      ,(caar clauses))
                    '(if ,(caar clauses)
                         (begin . ,(cdar clauses))
                         ,(loop (cdr clauses)))))
            (syntax-error "Wrong clause in COND" body))
        #!false))
  (loop (cdr body)))

(macro my-cond cond-expander)
```

Let us implement a macro BEGIN0 that implements a special form that takes a sequence of forms, evaluates them and returns the value of the first one.

```
(define (begin0-expander body)
  (define temp (gensym))
  (cond ((null? (cdr body))
         (syntax-error "Expression has too few subexpressions"
                       body))
        ((null? (cddr body))
         (cadr body))
        (else '((lambda (,temp) ,@(cddr body) ,temp)
                ,(cadr body)))))

(macro my-begin0 begin0-expander)

(define (and-expander form)
```

```
  (cond ((null? (cdr form)) #!true)
        ((null? (cddr form)) (cadr form))
        (else
          '(if ,(cadr form)
               ,(and-expander (cdr form))
               #!false))))

(macro my-and and-expander)

(define (or-expander form)
  (define temp (gensym))
  (cond ((null? (cdr form)) #!false)
        ((null? (cddr form)) (cadr form))
        (else
          '((lambda (,temp)
              (if ,temp
                  ,temp
                  ,(or-expander (cdr form))))
            ,(cadr form)))))

(macro my-or or-expander)
```

Problem:

Define macro WHEN that takes a predicate and any number of forms. It first evaluates the predicate and if it returns a true value evaluates the forms sequentially returning the value of the last form.

```
(define set-macro!
  (lambda (symbol function)
    (putprop symbol function 'pcs*macro)))

(define remove-macro!
  (lambda (symbol)
    (remprop symbol 'pcs*macro)))

(define macro-function
  (lambda (symbol)
    (getprop symbol 'pcs*macro)))

(define macroexpand-1
  (lambda (form)
    (cond ((symbol? form)
           (let ((x (macro-function form)))
             (if (pair? x)
                 (cdr x)
                 form)))
          ((and (pair? form)
                (symbol? (car form)))
           (let ((x (macro-function (car form))))
             (cond ((pair? x)
                    (cons (cdr x) (cdr form)))
                   ((procedure? x)
                    (x form))
                   (else form))))
          (else form))))

(define macroexpand
  (letrec
    ((loop
       (lambda (form)
         (let ((expansion (macroexpand-1 form)))
           (if (equal? form expansion)
               form
               (loop expansion))))))
    loop))

(define macroexpand-all
  (letrec
    ((loop
       (lambda (form)
         (let ((first-expansion (macroexpand form)))
           (if (and (pair? first-expansion)
                    (not (eq? (car first-expansion) 'quote)))
               (map loop first-expansion)
               first-expansion)))))
    loop))
```

```
(macro make-encapsulation
  (lambda (body)
    (let ((parameters (cadr body))
          (variables (caddr body))
          (local-procedures (cadddr body))
          (methods (car (cddddr body))))
      '(lambda ,parameters
         (let* ,variables
           (letrec ,(append local-procedures methods)
             (let ((list-of-methods
                     (list . ,(map (lambda (x)
                                      '(cons ',(car x) ,(car x)))
                                    methods))))
               (lambda (message)
                 (let ((method (assq message list-of-methods)))
                   (if (null? method)
                       (error
                         "no such method in this encapsulation: "
                          message)
                       (cdr method))))))))))

(macro old-use-methods
  (lambda (body)
    '(let ,(map (lambda (x)
                  (if (pair? x)
                      '(,(car x) (,(cadr body) ',(cadr x)))
                      '(,x (,(cadr body) ',x))))
                (caddr body))
       . ,(cdddr body))))

(macro use-methods
  (lambda (body)
    (define (clause-parser clause)
      (map (lambda (x)
             (if (pair? x)
                 '(,(car x) (,(car clause) ',(cadr x)))
                 '(,x (,(car clause) ',x))))
           (cadr clause)))
    '(let ,(map-append! clause-parser (cadr body))
       . ,(cddr body))))

(define (make-encapsulation-iterator encapsulation)
  (let ((pop! (encapsulation 'pop!))
        (empty? (encapsulation 'empty?)))
    (lambda (function)
      (do ()
          ((empty?))
          (function (pop!))))))
```

```
;;;==========
;;; Utilities
;;;==========

(define (vector-last v)
  (+ (vector-length v) 1))

(define (vector-swap! v i j)
  (let ((temp (vector-ref v i)))
    (vector-set! v i (vector-ref v j))
    (vector-set! v j temp)))

(define (vector-reverse! v)
  (do ((first 0 (1+ first))
       (last (vector-last v) (-1+ last)))
      ((>= first last) v)
      (vector-swap! v first last)))
```

```
;;;===========================================================
;;; Vector which only allows storage of improved values
;;;===========================================================

(define make-vector-with-predicate
  (make-encapsulation
    (n predicate)
    ((v (make-vector n 'empty)))
    ()
    ((set!? (lambda (index value)
         (cond ((or (eqv? (vector-ref v index) 'empty)
                    (predicate value (vector-ref v index)))
                (vector-set! v index value)
                #!TRUE)
               (else
                #!FALSE))))
     (ref (lambda (index) (vector-ref v index)))
     (values (lambda () v)))))
```

```
;;;=================================
;;; Deque implemented using a vector
;;;=================================

(define make-vector-deque
  (make-encapsulation
    (n)
    ((v (make-vector n))
     (number-of-nodes 0)
     (front 0)
     (rear 0)
     (last (-1+ n)))
    ((check-overflow
       (lambda () (if (full?) (error "deque overflow"))))
     (check-underflow
       (lambda () (if (empty?) (error "deque underflow"))))
     (increase-nodes! (lambda ()
                        (check-overflow)
                        (set! number-of-nodes
                              (1+ number-of-nodes))))
     (decrease-nodes! (lambda ()
                        (check-underflow)
                        (set! number-of-nodes
                              (-1+ number-of-nodes)))))
    ((full?
       (lambda () (= number-of-nodes n)))
     (empty?
       (lambda () (= number-of-nodes 0)))
     (in-rear! (lambda (value)
                 (increase-nodes!)
                 (vector-set! v rear value)
                 (set! rear (if (= rear last) 0 (1+ rear)))
                 *the-non-printing-object*))
     (in-front! (lambda (value)
                  (increase-nodes!)
                  (set! front (if (= front 0) last (-1+ front)))
                  (vector-set! v front value)
                  *the-non-printing-object*))
     (out-front! (lambda ()
                   (decrease-nodes!)
                   (let ((temp front))
                     (set! front (if (= front last)
                                     0
                                     (1+ front)))
                     (vector-ref v temp))))
     (out-rear! (lambda ()
                  (decrease-nodes!)
                  (set! rear (if (= rear 0) last (-1+ rear)))
                  (vector-ref v rear)))
     (peek-front (lambda ()
```

```
                    (check-underflow)
                    (vector-ref v front)))
     (peek-rear (lambda ()
                   (check-underflow)
                   (vector-ref v (if (= rear 0)
                                     last
                                     (-1+ rear)))))
     (length (lambda () number-of-nodes)))))
```

```
;;;==================================================
;;; Deque implemented with a vector-with-predicate
;;;==================================================

(define make-vector-deque-with-values
  (make-encapsulation
    (n predicate)
    ((v (make-vector-with-predicate n predicate))
     (queue (make-vector-deque n))
     (in-q (make-vector n 'never-was-in)))
    ((v-set!? (v 'set!?))
     (in-front! (queue 'in-front!))
     (in-rear! (queue 'in-rear!))
     (out-front! (queue 'out-front!)))
    ((push!?
       (lambda (index value)
         (cond ((v-set!? index value)
                (case (vector-ref in-q index)
                  ('never-was-in (in-rear! index))
                  ('was-in (in-front! index)))
                (vector-set! in-q index 'in)
                #!TRUE)
               (else #!FALSE))))
     (pop!
       (lambda ()
         (let ((value (out-front!)))
           (vector-set! in-q value 'was-in)
           value)))
     (v-ref (v 'ref))
     (empty? (queue 'empty?)))))
```

```
;;;=========================================
;;; Heap which keeps track of which elements
;;; of a fixed set are currently members.
;;;=========================================

(define make-heap-with-membership
  (make-encapsulation
    (n predicate)
    ((v (make-vector n))
     ;; v[i] = which index is in heap position i
     (used-to-be-in -1)
     (never-was-in -2)
     (member-v (make-vector n never-was-in))
     ;; mv[i] = where index i is in heap
     (fill-pointer -1))
    ((heap-set!
       (lambda (heap-position value)
       ;; put index value into position h-p in heap
         (vector-set! v heap-position value)
         (vector-set! member-v value heap-position)))
     (sift!
       (lambda (current step-function value predicate)
         (let ((next (step-function current)))
           (cond ((or (null? next)
                      (predicate value (vector-ref v next)))
                  (heap-set! current value))
                 (else (heap-set! current (vector-ref v next))
                       (sift! next step-function value
                   predicate))))))
     (heap-son
       (lambda (father)
         (let ((son (* 2 (1+ father))))
           (cond ((>= fill-pointer son)
                  (if (predicate (vector-ref v son)
                                 (vector-ref v (-1+ son)))
                      son
                      (-1+ son)))
                 ((= fill-pointer (-1+ son)) (-1+ son))
                 (else '())))))
     (heap-father
       (lambda (son)
         (if (>= 0 son) '() (quotient (-1+ son) 2))))
     (downheap!
       (lambda (father value)
         (sift! father heap-son value predicate)))
     (upheap!
       (lambda (son value)
         (sift! son heap-father value
                (lambda (x y) (predicate y x)))))
     (not-in?
```

```
        (lambda (index)
          (negative? index))))
     ((empty? (lambda () (= fill-pointer -1)))
      (push!
        (lambda (value)
          (let ((index (vector-ref member-v value)))
            (cond ((not-in? index)
                   (set! fill-pointer (1+ fill-pointer))
                   (upheap! fill-pointer value))
                  (else (upheap! index value))))))
      (pop!
        (lambda ()
          (let ((index (vector-ref v 0)))
            (vector-set! member-v index used-to-be-in)
            (set! fill-pointer (-1+ fill-pointer))
            (if (not (empty?))
                (downheap! 0 (vector-ref v (1+ fill-pointer))))
            index)))
      (unpopped?
        (lambda (index)
          (not (=? (vector-ref member-v index)
used-to-be-in))))))
```

```
;;;========================================
;;; Heap with membership implemented using
;;; a vector-with predicate and
;;; restricted so that once an index
;;; is popped, it cannot reenter the heap
;;;========================================

(define make-heap-with-membership-and-values
  (make-encapsulation
    (n predicate)
    ((v (make-vector-with-predicate n predicate))
     (ref (v 'ref))
     (heap (make-heap-with-membership
             n
             (lambda (x y) (predicate (ref x) (ref y))))))
    ((v-set!? (v 'set!?))
     (heap-push! (heap 'push!))
     (heap-unpopped? (heap 'unpopped?)))
    ((push!?
       (lambda (index value)
         (cond ((and (heap-unpopped? index)
                     (v-set!? index value))
                (heap-push! index)
                #!TRUE)
               (else #!FALSE))))
     (pop! (heap 'pop!))
     (v-ref ref)
     (empty? (heap 'empty?)))))
```

```
;;;
;;;   Make a scan-based algorithm.
;;;   This includes Bellman's, Dijkstra's and Prim's Algorithms.
;;;
;;;   Arguments:
;;;             make-data-structure
;;;             value-function
;;;             better?
;;;


(define (make-scan-based-algorithm
          make-data-structure value-function better?)
  (lambda (graph root)
    (let* ((encapsulation
             (make-data-structure
               ((graph 'number-of-nodes)) better?))
           (iterate-pop!
             (make-encapsulation-iterator encapsulation)))
      (use-methods
        ((graph
           (set-label! set-predecessor! second-node link-length
            for-each-node for-each-link-of-node number-of-nodes))
         (encapsulation
                      (push!? (label v-ref))))
        (for-each-node (lambda (x) (set-predecessor! x '())))
        (push!? root 0)
        (iterate-pop!
          (lambda (node)
            (for-each-link-of-node
              (lambda (link)
                (let ((new-node (second-node link)))
                  (when (push!?
                          new-node
                          (value-function (label node)
                               (link-length link)))
                        (set-predecessor! new-node link))))
              node)))
        (for-each-node
          (lambda (node) (set-label! node (label node)))))))))
```

```
;;;=========================================
;;;
;;;   Specific Algorithms
;;;
;;;=========================================

(define bellman
  (make-scan-based-algorithm
    make-vector-deque-with-values
;make-data-structure
    +                                                 ;value-function
    < ))                                              ;predicate

(define dijkstra
  (make-scan-based-algorithm
    make-heap-with-membership-and-values
;make-data-structure
    +                                                 ;value-function
    < ))                                              ;predicate

(define prim
  (make-scan-based-algorithm
    make-heap-with-membership-and-values
;make-data-structure
    (lambda (x y) y)                                  ;value-function
    < ))                                              ;predicate
```