10/10 WELLDONE!

ps2.scm


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS2
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; 2007/02/21
;;;
;;; Collaborators: '(iyzhang amdragon)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "ghelper.scm")
(load "generic-sequences.scm")

;;;
;;; Problem 1
;;;
;;; Implementations of the sequence operators are below.
;;;
;;; One notable implementation decision: many of the functions, such
;;; as map, are implemented in terms of streams. Many functions
;;; need to iterate over sequences in order, which needs to be done
;;; differently in order to be efficient for each data type: for
;;; example, we can reference each arbitrary index in a vector with
;;; sequence:ref, but for a list it's much faster to cdr down the
;;; list. So we define a sequence:stream operation that generates a
;;; stream from a sequence. The default implementation uses
;;; sequence:ref to fetch each element, but it's specialized for
;;; lists. Since it's a stream, we don't need to access elements until
;;; they become necessary (which makes this more efficient than
;;; coercing other sequence types into lists). It also makes it
;;; possible to mix different types of sequences as arguments without
;;; coercing (but that's Problem 2).

;;; Some helper functions

(define (assert x . msg)
  (if (not x) (apply error msg)))

(define (assert-equal a b)
  (assert (equal? a b)
          "Test failed"
          "Expected: "
          b
          " got: "
          a)
  #t)

(define (list-of? pred?)
  (lambda (lst)
    (and
     (list? lst)
     (for-all? lst pred?))))

(assert-equal ((list-of? string?) "a") #f)
(assert-equal ((list-of? string?) '("a" "b")) #t)
(assert-equal ((list-of? string?) '("a" 3)) #f)

;;; construct

(define generic:construct
  (make-generic-operator 2 #f))

(assign-operation generic:construct
                  (lambda (type items) items)
                  (is-exactly list?) any?)
(assign-operation generic:construct
                  (lambda (type items) (list->vector items))
                  (is-exactly vector?) any?)
(assign-operation generic:construct
                  (lambda (type items) (list->string items))
                  (is-exactly string?) any?)

(define (sequence:construct type . items)
  (generic:construct type items))

(assert-equal (sequence:construct list? 0 1 2 3 4) '(0 1 2 3 4))
(assert-equal (sequence:construct vector? 0 1 2 3 4) #(0 1 2 3 4))
(assert-equal (sequence:construct string? #\a #\b #\c) "abc")


;;; generate

(define sequence:generate
  (make-generic-operator 3 #f))

(define (range start end)
  (if (>= start end)
      '()
      (cons start (range (+ 1 start) end))))

(define (list-generate n fn)
  (map fn (range 0 n)))

(define (string-generate n fn)
  (apply string-append
   (list-generate n fn)))

(assign-operation sequence:generate
                  (lambda (type n fn) (string-generate n fn))
                  (is-exactly string?) exact-integer? any?)
(assign-operation sequence:generate
                  (lambda (type n fn) (list-generate n fn))
                  (is-exactly list?) exact-integer? any?)
(assign-operation sequence:generate
                  (lambda (type n fn) (make-initialized-vector n fn))
                  (is-exactly vector?) exact-integer? any?)

(assert-equal
 (sequence:generate list? 5 (lambda (x) x))
 '(0 1 2 3 4))
(assert-equal
 (sequence:generate string? 5 (lambda (x) (string (digit->char x))))
 "01234")
(assert-equal
 (sequence:generate vector? 5 (lambda (x) x))
 #(0 1 2 3 4))


;;; stream (convert sequence to stream)

(define (generic-sequence-stream sequence)
  (let ((len (sequence:size sequence)))
    (let loop ((i 0))
      (if (= i len)
          '()
          (cons-stream (sequence:ref sequence i)
                       (loop (+ i 1)))))))
```

2/2

Nice

Clever!

Nice!!

Nice

LIST? ... BUT OK BY CONSTRUCTION

~ iota [SEE MIT SCHEME REFMAN]

make-initialized-list

OK... OTHER FOLKS RESTRICTED fn TO RETURN A char, SO THEY JUST USED list->string.

procedure?

empty-stream

```
(define sequence:stream
  (make-generic-operator 1 generic-sequence-stream))

(assign-operation sequence:stream
                  list->stream
                  list?)
(assert-equal
 (stream->list
  (sequence:stream (sequence:generate vector? 5 (lambda (x) x))))
 (sequence:generate list? 5 (lambda (x) x)))
(assert-equal
 (stream->list
  (sequence:stream (sequence:generate list? 5 (lambda (x) x))))
 (sequence:generate list? 5 (lambda (x) x)))
(assert-equal
 (stream->list
  (sequence:stream
   (sequence:generate string? 5 (lambda (x) (string (digit->char x))))))
 (sequence:generate list? 5 (lambda (x) (digit->char x))))


;;; map

(define (generic-sequence-map type fn sequences)
  (apply sequence:construct
         type
         (let loop ((streams (map sequence:stream sequences)))
           (if (stream-null? (car streams))
               '()
               (let ((val (apply fn (map stream-car streams))))
                 (cons val
                       (loop (map stream-cdr streams))))))))


(define generic:map
  (make-generic-operator 3 generic-sequence-map))

(assign-operation generic:map
                  (lambda (type fn l) (apply map (cons fn l)))
                  (is-exactly list?) any? (list-of? list?))

(define (sequence:map fn . sequences)
  ;; XXX Should assert that sequences are of the same type, but see
  ;; Prob. 2
  (generic:map (sequence:type (car sequences))
               fn
               sequences))

(assert-equal
 (sequence:map (lambda (x) (+ 1 x))
               (sequence:generate list? 5 (lambda (x) x)))
 '(1 2 3 4 5))
(assert-equal
 (sequence:map (lambda (x) (+ 1 x))
               (sequence:generate vector? 5 (lambda (x) x)))
 #(1 2 3 4 5))
(assert-equal
 (sequence:map char-upcase "abc")
 "ABC")
(assert-equal
 (sequence:map * '(1 2 3 4 5) '(10 20 30 40 50))
 '(10 40 90 160 250))


;;; for-each

(define (generic-sequence-for-each type fn sequences)
  (let loop ((streams (map sequence:stream sequences)))
    (if (stream-null? (car streams))
        '()
        (let ((val (apply fn (map stream-car streams))))
          (cons val
                (loop (map stream-cdr streams)))))))


(define generic:for-each
  (make-generic-operator 3 generic-sequence-for-each))

(assign-operation generic:for-each
                  (lambda (type fn l) (apply for-each (cons fn l)))
                  (is-exactly list?) any? (list-of? list?))

(define (sequence:for-each fn . sequences)
  ;; XXX Should assert that sequences are of the same type, but see
  ;; Prob. 2
  (generic:for-each (sequence:type (car sequences))
                    fn
                    sequences))

(assert-equal
 (let ((lst '()))
   (sequence:for-each (lambda (x) (set! lst (cons x lst)))
                      '(1 2 3 4 5))
   lst)
 '(5 4 3 2 1))
(assert-equal
 (let ((lst '()))
   (sequence:for-each (lambda (x) (set! lst (cons x lst)))
                      #(1 2 3 4 5))
   lst)
 '(5 4 3 2 1))
(assert-equal
 (let ((lst '()))
   (sequence:for-each (lambda (x) (set! lst (cons x lst)))
                      "abc")
   lst)
 '(#\c #\b #\a))
(assert-equal
 (let ((lst '()))
   (sequence:for-each (lambda (x y) (set! lst (cons (+ x y) lst)))
                      '(1 2 3 4 5) '(10 20 30 40 50))
   lst)
 '(55 44 33 22 11))


;;; filter

(define (generic-sequence-filter seq pred)
  (apply sequence:construct
         (sequence:type seq)
         (let loop ((stream (sequence:stream seq)))
           (if (stream-null? stream)
               '()
               (if (pred (stream-car stream))
                   (cons (stream-car stream)
                         (loop (stream-cdr stream)))
                   (loop (stream-cdr stream)))))))

(define sequence:filter
  (make-generic-operator 2 generic-sequence-filter))
```

```
(assign-operation sequence:filter
                  (lambda (seq pred) (list-transform-positive seq pred))
                  list? any?)

(assert-equal
 (sequence:filter
  (sequence:generate list? 6 (lambda (x) x)) odd?)
 '(1 3 5))
(assert-equal
 (sequence:filter
  (sequence:generate vector? 6 (lambda (x) x)) odd?)
 #(1 3 5))
(assert-equal
 (sequence:filter
  (sequence:generate string? 5 (lambda (x) (string (digit->char x))))
  (lambda (x) (char<? x #\3)))
 "012")


;;; get-index

(define (generic-sequence-get-index seq pred)
  (let loop ((stream (sequence:stream seq))
             (i 0))
    (if (stream-null? stream)
        #f
        (if (pred (stream-car stream))
            i
            (loop (stream-cdr stream) (+ i 1))))))

(define sequence:get-index
  (make-generic-operator 2 generic-sequence-get-index))

(assert-equal
 (sequence:get-index
  (sequence:generate list? 6 (lambda (x) x)) odd?)
 1)
(assert-equal
 (sequence:get-index
  (sequence:generate list? 6 (lambda (x) x)) (lambda (x) (> x 10)))
 #f)
(assert-equal
 (sequence:get-index
  (sequence:generate vector? 6 (lambda (x) x)) odd?)
 1)
(assert-equal
 (sequence:get-index
  (sequence:generate string? 5 (lambda (x) (string (digit->char x))))
  (lambda (x) (char>? x #\3)))
 4)


;;; get-element

(define (generic-sequence-get-element seq pred)
  (let loop ((stream (sequence:stream seq)))
    (if (stream-null? stream)
        #f
        (if (pred (stream-car stream))
            (stream-car stream)
            (loop (stream-cdr stream))))))

(define sequence:get-element
  (make-generic-operator 2 generic-sequence-get-element))

(assert-equal
 (sequence:get-element
  (sequence:generate list? 6 (lambda (x) x)) even?)
 0)
(assert-equal
 (sequence:get-element
  (sequence:generate list? 6 (lambda (x) x)) (lambda (x) (> x 10)))
 #f)
(assert-equal
 (sequence:get-element
  (sequence:generate vector? 6 (lambda (x) x)) odd?)
 1)
(assert-equal
 (sequence:get-element
  (sequence:generate string? 5 (lambda (x) (string (digit->char x))))
  (lambda (x) (char>? x #\3)))
 #\4)


;;; fold-right

(define (generic-sequence-fold-right f init seq)
  (let loop ((stream (sequence:stream seq)))
    (if (stream-null? stream)
        init
        (f (stream-car stream)
           (loop (stream-cdr stream))))))

(define sequence:fold-right
  (make-generic-operator 3 generic-sequence-fold-right))

(assert-equal
 (sequence:fold-right + 0 '(1 2 3 4 5))
 15)
(assert-equal
 (sequence:fold-right (lambda (x r) (append r (list x))) '() '(1 2 3 4 5))
 '(5 4 3 2 1))
(assert-equal
 (sequence:fold-right + 0 #(1 2 3 4 5))
 15)
(assert-equal
 (sequence:fold-right (lambda (x r) (append r (list x))) '() #(1 2 3 4 5))
 '(5 4 3 2 1))
(assert-equal
 (sequence:fold-right (lambda (x r) (append r (list x))) '() "abcd")
 '(#\d #\c #\b #\a))


;;; fold-left

(define (generic-sequence-fold-left f init seq)
  (let loop ((stream (sequence:stream seq))
             (result init))
    (if (stream-null? stream)
        result
        (loop (stream-cdr stream)
              (f result (stream-car stream))))))

(define sequence:fold-left
  (make-generic-operator 3 generic-sequence-fold-left))

(assert-equal
 (sequence:fold-left + 0 '(1 2 3 4 5))
```

procedure?

For lists, could use find-matching-item

For lists, could use fold-right

For lists, could use fold-left

```
 15)
(assert-equal
 (sequence:fold-left (lambda (x y) (cons y x)) '() '(1 2 3 4 5))
 '(5 4 3 2 1))
(assert-equal
 (sequence:fold-left + 0 #(1 2 3 4 5))
 15)
(assert-equal
 (sequence:fold-left (lambda (x y) (cons y x)) '() #(1 2 3 4 5))
 '(5 4 3 2 1))
(assert-equal
 (sequence:fold-left (lambda (x y) (cons y x)) '() "abcd")
 '(#\d #\c #\b #\a))
```

2
2

WELL DONE

```
;;;
;;; Problem 2
;;;
;;; Several of the sequence functions take multiple sequences. We
;;; remove the restriction that the sequences need to be the same
;;; type:

;;; (sequence:append <sequence-1> ... <sequence-n>)
;;;    Returns a new sequence of the same type as sequence-1, formed
;;;    by concatenating the elements of the given sequences.  The size
;;;    of the new sequence is the sum of the sizes of the given
;;;    sequences. Sequences need not be of the same type.

;;; (sequence:map <function> <seq-1> ... <seq-n>)
;;;    Requires that the sequences given are of the same size, and
;;;    that the arity of the function is n.  The ith element of the
;;;    new sequence is the value of the function applied to the n ith
;;;    elements of the given sequences. Sequences need not be of the
;;;    same type.

;;; (sequence:for-each <procedure> <seq-1> ... <seq-n>)
;;;    Requires that the sequences given are of the same size, and
;;;    that the arity of the procedure is n.  Applies the procedure to
;;;    the n ith elements of the given sequences; discards the value.
;;;    This is done for effect. Sequences need not be of the same
;;;    type.

;;; Conveniently, implementing sequence:map and sequence:for-each does
;;; not require any modifications because of our stream-based
;;; implementation. We replace the implementation of sequence:append
;;; with one based on streams.

(define (generic-sequence-append type sequences)
  (apply sequence:construct
         type
         (apply append
                (map stream->list
                     (map sequence:stream sequences)))))

(define generic:append
  (make-generic-operator 2 generic-sequence-append))

(define (sequence:append . sequences)
  (generic:append (sequence:type (car sequences))
                  sequences))

(assert-equal (sequence:append '(1 2 3) '(4 5) '(6) '())
              '(1 2 3 4 5 6))
(assert-equal (sequence:append #(1 2 3) #(4 5) #(6) '())
              #(1 2 3 4 5 6))
(assert-equal (sequence:append "abc" "de" "f" "")
              "abcdef")
(assert-equal (sequence:append '(1 2 3) #(4 5) '(6) '())
              '(1 2 3 4 5 6))
(assert-equal (sequence:append #(1 2 3) '(4 5) '(6) '())
              #(1 2 3 4 5 6))

(assert-equal
 (sequence:map * '(1 2 3 4 5) #(10 20 30 40 50))
 '(10 40 90 160 250))
(assert-equal
 (sequence:map * #(1 2 3 4 5) '(10 20 30 40 50))
 #(10 40 90 160 250))
```

append-map!

Good

```
(assert-equal
 (let ((lst '()))
   (sequence:for-each (lambda (x y) (set! lst (cons (+ x y) lst)))
                      '(1 2 3 4 5) #(10 20 30 40 50))
   lst)
 '(55 44 33 22 11))
```

21/21

```
;;;
;;; Problem 3
;;;

;;; Many of the procedures above (sequence:map, sequence:for-each,
;;; sequence:append, ...) have variable arity, and their
;;; implementations take the following form: a (non-generic) wrapper
;;; function takes its arguments, turns most of them into a list, and
;;; calls a fixed-arity generic procedure. This is a somewhat awkward
;;; dance, and the common pattern suggests that a bit of syntactic
;;; sugar to build it might be useful.
;;;
;;; We could implement a function for building such wrappers, or more
;;; generally, add support for variable-arity functions into the
;;; generic dispatch system by adding support for predicates over the
;;; *entire* argument list, not separate predicates for each
;;; individual argument. The downside is that running more complicated
;;; predicates over the full argument list could be less efficient.
;;;
;;; To implement this, we'd need to modify assign-operation to take a
;;; single full-argument-list predicate rather than a list of
;;; per-argument predicates, and modify the operator function to test
;;; a single predicate against the full argument list rather than
;;; individual predicates against each argument.
```

Good.

```
;;;
;;; Problem 4
;;;
;;; a. This code is missing the test that returns false if it
;;; encounters an element in list-2 that is greater than than its
;;; corresponding element in list-1. As a result, it returns true if
;;; any element in list-1 is less than its corresponding element in
;;; list-2. Besides not matching the intuitive idea of how lists
;;; should be compared, it fails to define a total ordering: both
;;; (list<? '(1 2) '(2 1)) and (list<? '(2 1) '(1 2)) will return
;;; true. Chaos will ensue.
;;;
;;; b. Alyssa's explicit case analysis opts for defining the ordering
;;; in a single procedure rather than in n^2 generic dispatch
;;; entries. This seems worse for extensibility since adding a new
;;; type will require modifying the code for this procedure, rather
;;; than using the dispatch system to add new entries. In fact,
;;; however, the dispatch system isn't much better: adding a new type
;;; requires a dispatch entry for comparing it with any other type in
;;; the system (the n^2 problem). To achieve extensibility, what we
;;; really want to avoid is the need for implementors of a type to
;;; know about all other types in the system, and neither approach
;;; avoids this problem.
;;;
;;; An alternate approach (and, indeed, the one used below) is to
;;; define the total ordering of types in one place. Then a
;;; generic:less operator can have a default comparator that compares
;;; the types using the total ordering, and use dispatch entries to
;;; compare elements of the same type. This avoids the n^2 problem
;;; because all an implementor of a new type needs to do is add the
;;; type to the total ordering and provide a single generic:less for
;;; comparing two elements of that type.
;;;
;;; (We define the total ordering in a static list, but it's easy to
;;; imagine having functions to modify the list to dynamically add new
;;; types into it.)
;;;
;;; c. An implementation using the above technique follows:

;; null < Boolean < char < number < symbol < string < vector < list
(define type-ordering
  (list null? boolean? char? number? symbol? string? vector? list?))

(define (null<? a b) #f)
(define (boolean<? a b) (and (not a) b))

;; Alyssa's implementation, lifted straight from the text
(define (list<? list-1 list-2)
  (let ((len-1 (length list-1))
        (len-2 (length list-2)))
    (cond ((< len-1 len-2) #t)
          ((> len-1 len-2) #f)
          ;; Invariant:  equal lengths
          (else
           (let prefix<? ((list-1 list-1)
                          (list-2 list-2))
             (cond ((null? list-1) #f)  ; same
                   ((generic:less? (car list-1) (car list-2)) #t)
                   ((generic:less? (car list-2) (car list-1)) #f)
                   (else (prefix<? (cdr list-1) (cdr list-2)))))))))

(define (vector<? a b)
  (list<? (vector->list a) (vector->list b)))
```

```
;; Compare two elements according to type. Return an error if they
;; have the same type. (This condition is included for safety, as it
;; ensures that generic:less? can't be used if a predicate for
;; comparing two elements of a particular type isn't added. It could
;; be removed so that generic:less? would always return #f instead in
;; this case.)
(define (type<? a b)
  (let ((a-ind (sequence:get-index type-ordering
                                   (lambda (type-pred) (type-pred a))))
        (b-ind (sequence:get-index type-ordering
                                   (lambda (type-pred) (type-pred b)))))
    (cond ((not a-ind) (error "Unknown type for" a))
          ((not b-ind) (error "Unknown type for" b))
          ((eqv? a-ind b-ind)
           (error "Attempting to compare elements of the same type" a b))
          (else (< a-ind b-ind)))))

(define generic:less?
  (make-generic-operator 2 type<?))

(assign-operation generic:less? null<? null? null?)
(assign-operation generic:less? boolean<? boolean? boolean?)
(assign-operation generic:less? char<? char? char?)
(assign-operation generic:less? < number? number?)
(assign-operation generic:less? symbol<? symbol? symbol?)
(assign-operation generic:less? string<? string? string?)
(assign-operation generic:less? vector<? vector? vector?)
(assign-operation generic:less? list<? list? list?)

(assert-equal (generic:less? '() '()) #f)
(assert-equal (generic:less? #f #f) #f)
(assert-equal (generic:less? #t #f) #f)
(assert-equal (generic:less? #f #t) #t)
(assert-equal (generic:less? #\a #\b) #t)
(assert-equal (generic:less? #\a #\a) #f)
(assert-equal (generic:less? #\b #\a) #f)
(assert-equal (generic:less? 1 2) #t)
(assert-equal (generic:less? 2 1) #f)
(assert-equal (generic:less? 1 1) #f)
(assert-equal (generic:less? 'a 'b) #t)
(assert-equal (generic:less? 'b 'a) #f)
(assert-equal (generic:less? 'a 'a) #f)
(assert-equal (generic:less? "aa" "ab") #t)
(assert-equal (generic:less? "aa" "aa") #f)
(assert-equal (generic:less? "ab" "aa") #f)
(assert-equal (generic:less? #(1 2) #(2 1)) #t)
(assert-equal (generic:less? #(1 2) #(1 2)) #f)
(assert-equal (generic:less? #(2 1) #(1 2)) #f)
(assert-equal (generic:less? '(1 2) '(2 1)) #t)
(assert-equal (generic:less? '(1 2) '(1 2)) #f)
(assert-equal (generic:less? '(2 1) '(1 2)) #f)

(assert-equal (generic:less? '() '()) #f)
(assert-equal (generic:less? '() #f) #t)
(assert-equal (generic:less? '() #\a) #t)
(assert-equal (generic:less? '() 1) #t)
(assert-equal (generic:less? '() 'a) #t)
(assert-equal (generic:less? '() "a") #t)
(assert-equal (generic:less? '() #(1)) #t)
(assert-equal (generic:less? '() '(1)) #t)
(assert-equal (generic:less? #f '()) #f)
(assert-equal (generic:less? #f #f) #f)
(assert-equal (generic:less? #f #\a) #t)
(assert-equal (generic:less? #f 1) #t)
(assert-equal (generic:less? #f 'a) #t)
```

```
(assert-equal (generic:less? #f "a") #t)
(assert-equal (generic:less? #f #(1)) #t)
(assert-equal (generic:less? #f '(1)) #t)
(assert-equal (generic:less? #\a '()) #f)
(assert-equal (generic:less? #\a #f) #f)
(assert-equal (generic:less? #\a #\a) #f)
(assert-equal (generic:less? #\a 1) #t)
(assert-equal (generic:less? #\a 'a) #t)
(assert-equal (generic:less? #\a "a") #t)
(assert-equal (generic:less? #\a #(1)) #t)
(assert-equal (generic:less? #\a '(1)) #t)
(assert-equal (generic:less? 1 '()) #f)
(assert-equal (generic:less? 1 #f) #f)
(assert-equal (generic:less? 1 #\a) #f)
(assert-equal (generic:less? 1 1) #f)
(assert-equal (generic:less? 1 'a) #t)
(assert-equal (generic:less? 1 "a") #t)
(assert-equal (generic:less? 1 #(1)) #t)
(assert-equal (generic:less? 1 '(1)) #t)
(assert-equal (generic:less? 'a '()) #f)
(assert-equal (generic:less? 'a #f) #f)
(assert-equal (generic:less? 'a #\a) #f)
(assert-equal (generic:less? 'a 1) #f)
(assert-equal (generic:less? 'a 'a) #f)
(assert-equal (generic:less? 'a "a") #t)
(assert-equal (generic:less? 'a #(1)) #t)
(assert-equal (generic:less? 'a '(1)) #t)
(assert-equal (generic:less? "a" '()) #f)
(assert-equal (generic:less? "a" #f) #f)
(assert-equal (generic:less? "a" #\a) #f)
(assert-equal (generic:less? "a" 1) #f)
(assert-equal (generic:less? "a" 'a) #f)
(assert-equal (generic:less? "a" "a") #f)
(assert-equal (generic:less? "a" #(1)) #t)
(assert-equal (generic:less? "a" '(1)) #t)
(assert-equal (generic:less? #(1) '()) #f)
(assert-equal (generic:less? #(1) #f) #f)
(assert-equal (generic:less? #(1) #\a) #f)
(assert-equal (generic:less? #(1) 1) #f)
(assert-equal (generic:less? #(1) 'a) #f)
(assert-equal (generic:less? #(1) "a") #f)
(assert-equal (generic:less? #(1) #(1)) #f)
(assert-equal (generic:less? #(1) '(1)) #t)
(assert-equal (generic:less? '(1) '()) #f)
(assert-equal (generic:less? '(1) #f) #f)
(assert-equal (generic:less? '(1) #\a) #f)
(assert-equal (generic:less? '(1) 1) #f)
(assert-equal (generic:less? '(1) 'a) #f)
(assert-equal (generic:less? '(1) "a") #f)
(assert-equal (generic:less? '(1) #(1)) #f)
(assert-equal (generic:less? '(1) '(1)) #f)

;; Remove duplicates from a *sorted* list
(define (remove-duplicates l)
  (fold-right (lambda (x y)
                (cond ((null? y) (cons x y))
                      ((equal? x (car y)) y)
                      (else (cons x y))))
              '() l))
(assert-equal (remove-duplicates '(1 2 3)) '(1 2 3))
(assert-equal (remove-duplicates '(1 2 2 3)) '(1 2 3))
(assert-equal (remove-duplicates '(1 2 2 3 3 3 3 3)) '(1 2 3))
(assert-equal (remove-duplicates '()) '())

(define (generic:sequence->set seq)
  (remove-duplicates
   (sort
    (stream->list (sequence:stream seq))
    generic:less?)))

(assert-equal (generic:sequence->set '(1 2 3)) '(1 2 3))
(assert-equal (generic:sequence->set '(3 4 1 3 2 1)) '(1 2 3 4))
(assert-equal (generic:sequence->set #(3 4 1 3 2 1)) '(1 2 3 4))
(assert-equal (generic:sequence->set "dbabcd") '(#\a #\b #\c #\d))

;; Set operations
(define set:equal? equal?)
(assert-equal (set:equal? (generic:sequence->set '(1 2 3))
                          (generic:sequence->set '(1 3 2))) #t)
(assert-equal (set:equal? (generic:sequence->set '(1 2 3))
                          (generic:sequence->set '(1 3 2 4))) #f)

;; O(n lg n) implementation purely because I'm lazy; could be made O(n)
(define (set:union a b)
  (generic:sequence->set (append a b)))

(assert-equal (set:equal? (set:union (generic:sequence->set '(1 2 3))
                                     (generic:sequence->set '(3 2 4)))
                          (generic:sequence->set '(1 2 3 4))) #t)

(define (set:intersection a b)
  (cond ((null? a) '())
        ((null? b) '())
        ((equal? (car a) (car b))
         (cons (car a) (set:intersection (cdr a) (cdr b))))
        ((generic:less? (car a) (car b))
         (set:intersection (cdr a) b))
        (else (set:intersection a (cdr b)))))

(assert-equal (set:equal? (set:intersection (generic:sequence->set '(1 2 3))
                                            (generic:sequence->set '(3 2 4)))
                          (generic:sequence->set '(2 3))) #t)

(define (set:difference a b)
  (cond ((null? a) '())
        ((null? b) a)
        ((equal? (car a) (car b))
         (set:difference (cdr a) (cdr b)))
        ((generic:less? (car a) (car b))
         (cons (car a ) (set:difference (cdr a) b)))
        (else (set:difference a (cdr b)))))

(assert-equal (set:equal? (set:difference
                           (generic:sequence->set '(5 1 2 3))
                           (generic:sequence->set '(3 4)))
                          (generic:sequence->set '(1 2 5))) #t)

(define (set:strict-subset? a b)
  (cond ((null? b) #f)
        ((null? a) #t)
        ((equal? (car a) (car b))
         (set:strict-subset? (cdr a) (cdr b)))
        ((generic:less? (car a) (car b)) #f)
        (else (set:strict-subset? a (cdr b)))))

(assert-equal (set:strict-subset? (generic:sequence->set '(5 1 2 3))
                                  (generic:sequence->set '(3 4))) #f)
(assert-equal (set:strict-subset? (generic:sequence->set '(1 2 3))
                                  (generic:sequence->set '(3 4 1 2 5))) #t)
(assert-equal (set:strict-subset? (generic:sequence->set '(1 2 3))
```

YAHOO!

EXCELLENT!

OK

OK SO LONG AS YOU CONFESS ☺

```
(generic:sequence->set '(3 2 1))) #f)

;;; d) If we hadn't followed Alyssa's suggestion that sets be
;;; represented as sorted, non-redundant lists, and instead had
;;; allowed them to be any sequence (possibly unsorted or redundant),
;;; generic:sequence->set would have been completely trivial, but
;;; would have pushed all of the work into the various set
;;; operations. These operations would either have to sort the
;;; sequence and remove duplicates themselves, giving a O(n lg n)
;;; runtime and silly implementation complexity, or would need O(n^2)
;;; algorithms that involved repeatedly iterating over one list to
;;; check which items are in it. In contrast, our set operations can
;;; have linear cost, so we have both improved code simplicity and
;;; faster runtime.
```

EXACTLY!

2/2

WELL DONE.

```
;;;
;;; Problem 5
;;;
;;;
;;; The choice between dispatching on predicates and dispatching on
;;; tags is a tradeoff between flexibility and speed. To dispatch
;;; based on predicates requires testing the arguments against the
;;; predicates until a match is found. This can be difficult, since
;;; there may be many predicates to test against. More significantly,
;;; however, the tests are arbitrary code, requiring at least the
;;; overhead of a function call and at worst never terminating. It's
;;; not unreasonable that a predicate would need to examine the
;;; contents of its arguments (such as testing whether each element in
;;; a list is a character), which is a linear-time operation, and even
;;; more complex predicates aren't inconceivable. Indeed, if we
;;; support variable-arity operators, complex predicates can become
;;; common.
;;;
;;; To dispatch based on tags, we simply need to examine the tags of
;;; each argument and compare them to entries in a dispatch
;;; table. This requires only a single equality check against the tags
;;; of possible handlers. Using a hash table or other data structures,
;;; it's possible to reduce the number of comparisons that need to be
;;; done in the average case --- but the same is true of predicate
;;; dispatch, since we may be able to structure the data to limit the
;;; number of predicates that need to be tested on average. For
;;; example, a tree with the most common predicates near the top means
;;; that the less common predicates do not need to be tested unless
;;; they are necessary. But predicate testing remains more expensive
;;; than tag checking.
;;;
;;; Predicate dispatch does provide more general operations, since we
;;; can specify predicates more complex than types. For example, we
;;; can define an operator that works one way for positive numbers and
;;; a different way for negative numbers. We couldn't do this with
;;; tagged data unless we had a separate tag for positive and negative
;;; numbers, which would mean we'd have to anticipate the need to
;;; handle positive and negative numbers differently at the time we
;;; created the data, rather than at the time we created each
;;; operation for the operator. (Also, this sort of tagging would
;;; quickly grow hopelessly cumbersome as we add more predicates ---
;;; we'd have to tag numbers as being positive-odd-nonprime numbers,
;;; etc.)
```

WELL ARGUED. SEE ALSO THE RECOMMENDED READING BY M. ERNST.