MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2013
Problem Set 2

Issued: Wed. 13 Feb. 2013 **Due: Wed. 20 Feb. 2013**

Reading:
SICP sections 2.4 and 2.5
(Tagged data, Data-directed programming, Generic Operations)

If you are really interested in generic dispatch see the paper by Ernst, et al. Do not obsess over the formal semantics, what is really interesting here is the way predicate dispatch can be used to subsume other kinds of dispatch.
http://pag.csail.mit.edu/~mernst/pubs/dispatching-ecoop98-abstract.html

Code: ghelper.scm, generic-specs.scm, generic-sequences.scm, load.scm

Documentation:
The MIT/GNU Scheme documentation
online at http://www.gnu.org/software/mit-scheme/

Generic Operations

In this problem set we will explore a variety of methods we can use for implementing and exploiting generic operations.

The procedures in the file ghelper.scm are an elegant mechanism for implementing generic-operator dispatch, where the handlers for the generic operators are specified by the predicates that the arguments satisfy. The file generic-specs.scm is an informal programmer's specification of generic operations that can be defined over a variety of ordered linear data structures, such as lists, vectors, and strings. The file generic-sequences.scm is a beginning implementation of the generic operators specified in generic-specs.scm.

To load ghelper and generic-sequences incant (load "load") after (cd "your code directory") at a Scheme read-eval-print loop.

Problem 2.1:

Complete the implementation started in generic-sequences.scm to match the specifications in generic-specs.scm. Demonstrate that each of your generic operators works as specified, by showing examples. You should insert your tests as comments in the code you hand in.

Notice that the types in the underlying Scheme are not uniformly specified, so this is not entirely trivial: in our seed file, for example, we had to define vector-null?, list-set!, and vector-append just to fill things out a bit.

Operations like sequence:append can be extended to allow the combination of unlike sequences. For example, we might expect to be able to write

```
(sequence:append (list 'a 'b 'c) (vector 'd 'e 'f))
```

and get back the list (a b c d e f), assuming that we want a sequence of the first argument type to be the sequence type of the result.

One way to implement this sort of thing is to write specific handlers for all the combinations of types we might want. This may be a large problem. However, the problem can be mitigated by using coercions, such as vector->list, list->vector, etc. The cost of doing the coercions is the construction of a new intermediate data structure that is not needed in the result. This may or may not be important, depending on the application. With coercions, we make up and use new combinators to help construct the generic operator entries:

```
(define (compose-1st-arg f g)
  (lambda (x y) (f (g x) y)))

(define (compose-2nd-arg f g)
  (lambda (x y) (f x (g y))))
```

Using these we can write such things as:

```
(defhandler generic:binary-append
  (compose-2nd-arg vector-append list->vector)
  vector? list?)

(defhandler generic:binary-append
  (compose-2nd-arg append vector->list)
  list? vector?)
```

Problem 2.2:

Examine the generic specifications. What generalizations that mix combinations of sequence types may be useful? Amend the specification document so as to include the generalization. (Turn in the amended specification sheet with your changes clearly indicated.) Amend your implementation to make these generalizations.

Some of the coercions that you may need are provided by Scheme, but others may need to be written, such as vector->string. (Consult the online MIT/GNU Scheme reference manual to see what is and is not provided.)

The generic procedure sequence:append also illustrates the problem that we must arbitrarily select the type of the output value for each generic procedure. However, the type required may be different depending upon the way that value will be used. For example, complex numbers are best expressed in polar form if we intend to multiply them, whereas they are best expressed in rectangular form if we intend to add them. Automagically knowing the right form is itself an interesting and complicated problem (that we may address later in this subject) but it may be useful, as an intermediate solution, to allow the user to specify the desired output type for each call site. This provides an advantage over explicit after-the-fact coercion of the output by the user, because it gives the dispatch mechanism the ability to select what may be the best way to accomplish the desired result. For example, it may be much more expensive to develop the answer the wrong format than to develop it in the desired format.

A user might specify the type of the result of the sequence:append operation by giving an (optional) first argument that is a type predicate as follows:

If a vector is desired the user could specify

```
(sequence:append vector? (list 'a 'b 'c) (vector 'd 'e 'f))
```

if a list is desired the user could specify

```
(sequence:append list? (list 'a 'b 'c) (vector 'd 'e 'f))
```

the system default is specified by omitting the type predicate

```
(sequence:append (list 'a 'b 'c) (vector 'd 'e 'f))
```

Problem 2.3

Is this a good idea? (Please state and argue your opinion.) Are there disadvantages to this syntactic scheme? Do you have a better idea?

What changes would you have to make in the ghelper.scm file to implement some form of target selection (either the suggestion above or your better idea)? For example, how would make-generic-operator have to change? defhandler? Implement your changes and test them.

The code for sequence:append illustrates another interesting problem. Our generic dispatch program does not allow us to make generic operations with unspecified arity -- that take many arguments -- such as addition. We programmed around that restriction by defining a binary generic operation and then using a folding reduction (fold-right) to extend the binary operation to take an arbitrary number of arguments. However, the folding reduction needs to know the null sequence of the type being constructed. Alternatively, we could have extended the generic dispatch to allow creation of procedures with unspecified arity. This would allow us to move the folding to the type-specific procedures rather than make it a wrapper around the binary generic procedure.

Problem 2.4

Is this a good idea? (Please state and argue your opinion.)
How does this interact with Problem 2.3 above?

Assuming that we want to do this, what changes would you have to make in the ghelper.scm file? For example, how would make-generic-operator have to change? defhandler? We do not want you to actually implement these changes, just think about what would have to be done and informally describe your conclusions.

Ben Bitdiddle is pleased with our generic sequences but notes that, beyond generic N-tuples, it is useful also to have generic sets. He proposes that we further extend our language with:

(generic:sequence->set <sequence>)
Returns a list corresponding to <sequence> with no duplicates.
Duplication is determined using EQUAL? (not EQ? nor EQV?).

The remaining traditional set operations are straightforward:

```
(set:equal?         <set-1> <set-2>)
(set:union          <set-1> <set-2>)
(set:intersection   <set-1> <set-2>)
(set:difference     <set-1> <set-2>) - E.g. {A,B,C}-{9,B,D}={A,C}
(set:strict-subset? <set-1> <set-2>)
```

Alyssa P. Hacker is quick to point out that an efficient way to implement sets is as sorted, irredundant lists. She adds, ‘‘Of course, this would require a generic:less? predicate to induce a total order on the potential set elements.’’

To that end, Alyssa proposes the following ordering on types of objects:

null < Boolean < char < number < symbol < string < vector < list

She notes that MIT Scheme already provides handy implementations of each of: char<?, <, symbol<? and string<?. Adding that null<? and boolean<? are straightforward to define and that vector<? can just cheat and resort to list<? (for now), she cautions that list<?, on the other hand, must take special care to ensure that:

(generic:less? x y) implies (not (generic:less? y x))

...in order to be well defined (and, thus, well behaved), although list<? can, of course, leverage generic:less? in any recursive subexpression predications.

Louis Reasoner, ignoring this advice, proposes the following implementation of list<?:

```
(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
          ((null? list-1) #f)           ; same
          (else
           (or (generic:less? (car list-1) (car list-2))
               (generic:less? (cdr list-1) (cdr list-2)))))))
```

Alyssa counters that the following is more appropriate:

```
(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)))))))))
```

As a parting shot, Alyssa also advises that entering N^2 items into the generic dispatch table can be avoided by just defining generic:less? outright, as per:

```
(define (generic:less? x y)
  (cond ((null?    x) (if (null?    y) (null<?    x y) #t))
        ((null?    y) #f)
        ((boolean? x) (if (boolean? y) (boolean<? x y) #t))
        ((boolean? y) #f)
        ...
        (else (error "Unrecognized data type" x))))
```

Problem 2.5:

A. What's wrong with Louis' implementation of the list<? predicate? Give a simple example and a brief explanation of what problems this would cause if it were used in generic:less? to sort sets.

B. Briefly critique Alyssa's suggesting for implementing generic:less? as an explicit case analysis versus using the dispatch table.

C. Implement and demonstrate a generic:less? operation using Alyssa's total ordering of data types (and her list<? code), but using the generic dispatch mechanism instead of an explicit conditional shown above.

The system for implementing generic operations that we have looked at so far in this problem set is extremely general and flexible: the dispatch to a handler is based on arbitrary predicates applied to the arguments. Most generic operation systems are more constrained, in that the arguments are presumed to have types that are determined either statically by some declaration mechanism or by a type tag that is associated with the argument data. For example, in the SICP readings for this problem set, the data is tagged and the dispatch is based on these tags. Such a tagged-data system has important advantages of efficiency, but it gives up some flexibility.

Problem 2.6:

How much does dispatch on predicates cost? What is the fundamental efficiency problem here? Imagine that we have a system with tagged data, but that we test for the tags with predicates. What can be done with the data tags that can eliminate much of the work of the predicate-based system?

On the other hand, what do we give up in a more conventional system, such as the one outlined in SICP, by contrast to the predicate-based system? What is an example of lost flexibility?

Under what circumstances could you build a generic dispatch system with zero runtime overhead? What flexibility would you be giving up? What about a system with very little or constant-time overhead?

Write a few clear paragraphs expounding on these ideas. Try to separate accident from essence. (Some aspects of a system are consequences of accidental choices--ones that could easily be changed--such as the use of a hash table rather than an association list. Other aspects are essential in that no local modifications can significantly change the behavior.)

```
;;;;            Generic sequence operations
;;;                   generic-specs.scm

;;; There are many kinds of data that can be used to represent sequences:
;;;     examples include strings, lists, and vectors.

;;; There are operations that can be defined for all sequence types.

;;;                     Constructing
;;;
;;; (sequence:construct <sequence-type> <item-1> ... <item-n>)
;;;    Constructs a new sequence of the given type and of size n with
;;;    the given elements: item-1 ... item-n

;;; (sequence:null <sequence-type>)
;;;    Produces the null sequence of the given type


;;;                      Selecting
;;;
;;; (sequence:ref <sequence> <i>)
;;;    Returns the ith element of the sequence.  We use zero-based
;;;    indexing, so for a sequence of length n the ith item is
;;;    referenced by (sequence:ref <sequence> <i-1>).

;;; (sequence:size <sequence>)
;;;    Returns the number of elements in the sequence.

;;; (sequence:type <sequence>)
;;;    Returns the predicate defining the type of the sequence given.


;;;                      Testing
;;;
;;; (sequence:null? <sequence>)
;;;    Returns #t if the sequence is null, otherwise returns #f.

;;; (sequence:equal? <sequence-1> <sequence-2>)
;;;    Returns #t if the sequences are of the same type and have equal
;;;    elements in the same order, otherwise returns #f.


;;;                      Mutation
;;;
;;; Some sequences are immutable, while others can be changed.
;;;
;;; For those that can be modified we can change an element:
;;;
;;; (sequence:set! <sequence> <i> <v>)
;;;    Sets the ith element of the sequence to v.
```

```
;;;                  Cutting and Pasting
;;;
;;;  (sequence:subsequence <sequence> <start> <end>)
;;;    The arguments start and end must be exact integers such that
;;;       0 <= start <= end <= (sequence:size <sequence>).
;;;    Returns a new sequence of the same type as the given sequence,
;;;    of size end-start with elements selected from the given sequence.
;;;    The new sequence starts with the element of the given sequence
;;;    referenced by start.  It ends with the element of the given
;;;    sequence referenced by end-1.

;;; (sequence:append <sequence-1> ... <sequence-n>)
;;;    Requires that the sequences are all of the same type.  Returns
;;;    a new sequence of the type, 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.

;;;                      Iterators
;;;
;;; (sequence:generate <sequence-type> <n> <function>)
;;;    Makes a new sequence of the given sequence type, of size n.
;;;    The ith element of the new sequence is the value of the
;;;    function at the index i.

;;; (sequence:map <function> <seq-1> ... <seq-n>)
;;;    Requires that the sequences given are of the same size and
;;;    type, 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.

;;; (sequence:for-each <procedure> <seq-1> ... <seq-n>)
;;;    Requires that the sequences given are of the same size and
;;;    type, 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.

;;;                  Filtration and Search
;;;
;;; (sequence:filter <sequence> <predicate>)
;;;    Returns a new sequence with exactly those elements of the given
;;;    sequence for which the predicate is true (does not return #f).
;;;
;;; (sequence:get-index <sequence> <predicate>)
;;;    Returns the index of the first element of the sequence that
;;;    satisfies the predicate.  Returns #f if no element of the
;;;    sequence satisfies the predicate.
;;;
;;; (sequence:get-element <sequence> <predicate>)
;;;    Returns the first element of the sequence that satisfies the
;;;    predicate.  Returns #f if no element of the sequence satisfies
;;;    the predicate.
```

```
;;;                    Accumulation
;;;
;;; (sequence:fold-right <function> <initial> <sequence>)
;;;    Returns the result of applying the given binary function,
;;;    from the right, starting with the initial value.
;;;    For example,
;;;      (sequence:fold-right list 'end '(a b c))
;;;           => (a (b (c end)))

;;;
;;; (sequence:fold-left <function> <initial> <sequence>)
;;;    Returns the result of applying the given binary function,
;;;    starting with the initial value, from the left.
;;;    For example,
;;;      (sequence:fold-left list 'start '(a b c))
;;;           => (((start a) b) c)
```

```
;;;;    Generic sequence operator definitions
;;;             generic-sequences.scm

;;; First we declare the operators we want to be generic.
;;;  Each declaration specifies the arity (number of arguments).
;;;  It may specify a name and a default operation, if necessary.

(define sequence:null
  (make-generic-operator 1))


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

(define sequence:size
  (make-generic-operator 1))

(define sequence:type
  (make-generic-operator 1))

(define sequence:null?
  (make-generic-operator 1))

(define sequence:equal?
  (make-generic-operator 2))

(define sequence:set!
  (make-generic-operator 3))

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


;;; sequence:append takes multiple arguments.  It is defined in terms
;;; of a binary generic append that takes a sequence and a list of
;;; sequences.

(define (sequence:append . sequences)
  (if (null? sequences)
      (error "Need at least one sequence for append"))
  (let ((type? (sequence:type (car sequences))))
    (if (not (for-all? (cdr sequences) type?))
        (error "All sequences for append must be of the same type"
               sequences))
    (fold-right generic:binary-append (sequence:null type?) sequences)))

(define generic:binary-append (make-generic-operator 2))
```

```
;;; Implementations of the generic operators.

(define (any? x) #t)
(define (constant val) (lambda (x) val))
(define (is-exactly val) (lambda (x) (eq? x val)))

(defhandler sequence:null (constant "")    (is-exactly string?))
(defhandler sequence:null (constant '())   (is-exactly list?))
(defhandler sequence:null (constant #())   (is-exactly vector?))

(defhandler sequence:ref string-ref string? exact-nonnegative-integer?)
(defhandler sequence:ref list-ref   list?   exact-nonnegative-integer?)
(defhandler sequence:ref vector-ref vector? exact-nonnegative-integer?)

(defhandler sequence:size string-length     string?)
(defhandler sequence:size length            list?)
(defhandler sequence:size vector-length     vector?)

(defhandler sequence:type (constant string?)     string?)
(defhandler sequence:type (constant list?)       list?)
(defhandler sequence:type (constant vector?)     vector?)


(define (vector-null? v) (= (vector-length v) 0))

(defhandler sequence:null? string-null?     string?)
(defhandler sequence:null? null?            list?)
(defhandler sequence:null? vector-null?     vector?)


;;; To assign to the ith element of a list:

(define (list-set! list i val)
  (cond ((null? list)
         (error "List does not have enough elements" i))
        ((= i 0) (set-car! list val))
        (else (list-set! (cdr list) (- i 1) val))))

(defhandler sequence:set! string-set!
                  string? exact-nonnegative-integer? any?)
(defhandler sequence:set! list-set!
                  list?   exact-nonnegative-integer? any?)
(defhandler sequence:set! vector-set!
                  vector? exact-nonnegative-integer? any?)
```

```
(defhandler sequence:subsequence substring
          string? exact-nonnegative-integer? exact-nonnegative-integer?)

(defhandler sequence:subsequence sublist
          list? exact-nonnegative-integer?   exact-nonnegative-integer?)

(defhandler sequence:subsequence subvector
          vector? exact-nonnegative-integer? exact-nonnegative-integer?)


(define (vector-append v1 v2)
  (let ((n1 (vector-length v1))
        (n2 (vector-length v2)))
    (make-initialized-vector (+ n1 n2)
                             (lambda (i)
                               (if (< i n1)
                                   (vector-ref v1 i)
                                   (vector-ref v2 (- i n1)))))))

(defhandler generic:binary-append string-append  string? string?)
(defhandler generic:binary-append append         list?   list?)
(defhandler generic:binary-append vector-append  vector? vector?)
```

```
;;;;           Most General Generic-Operator Dispatch

(declare (usual-integrations))

;;; Generic-operator dispatch is implemented here by a discrimination
;;; list, where the arguments passed to the operator are examined by
;;; predicates that are supplied at the point of attachment of a
;;; handler (by DEFHANDLER).

;;; To be the correct branch all arguments must be accepted by
;;; the branch predicates, so this makes it necessary to
;;; backtrack to find another branch where the first argument
;;; is accepted if the second argument is rejected.  Here
;;; backtracking is implemented by OR.

(define (make-generic-operator arity #!optional name default-operation)
  (let ((record (make-operator-record arity)))
    (define (operator . arguments)
      (if (not (= (length arguments) arity))
          (error "Wrong number of arguments for generic operator"
                 (if (default-object? name) operator name)
                 arity arguments))
      (apply (or (let per-arg
                     ((tree (operator-record-tree record))
                      (args arguments))
                   (let per-pred ((tree tree))
                     (and (pair? tree)
                          (if ((caar tree) (car args))
                              (if (pair? (cdr args))
                                  (or (per-arg (cdar tree) (cdr args))
                                      (per-pred (cdr tree)))
                                  (cdar tree))
                              (per-pred (cdr tree))))))
                 (if (default-object? default-operation)
                     (lambda args
                       (error "No applicable methods for generic operator"
                              (if (default-object? name) operator name)
                              args))
                     default-operation))
             arguments))

    (hash-table/put! *generic-operator-table* operator record)
    operator))

(define *generic-operator-table*
  (make-eq-hash-table))

(define (get-operator-record operator)
  (hash-table/get *generic-operator-table* operator #f))
```

```
(define (make-operator-record arity) (cons arity '()))
(define (operator-record-arity record) (car record))
(define (operator-record-tree record) (cdr record))
(define (set-operator-record-tree! record tree) (set-cdr! record tree))

(define (defhandler operator handler . argument-predicates)
  (let ((record
         (let ((record (hash-table/get *generic-operator-table* operator #f))
               (arity (length argument-predicates)))
           (if record
               (begin
                 (if (not (= arity (operator-record-arity record)))
                     (error "Incorrect operator arity:" operator))
                 record)
               (error "Operator not known" operator)))))
    (set-operator-record-tree! record
      (bind-in-tree argument-predicates
                    handler
                    (operator-record-tree record))))
  operator)

;;; An alias used in some older code
(define assign-operation defhandler)


(define (bind-in-tree keys handler tree)
  (let loop ((keys keys) (tree tree))
    (let ((p.v (assq (car keys) tree)))
      (if (pair? (cdr keys))
          (if p.v
              (begin
                (set-cdr! p.v
                          (loop (cdr keys) (cdr p.v)))
                tree)
              (cons (cons (car keys)
                          (loop (cdr keys) '()))
                    tree))
          (if p.v
              (begin
                (warn "Replacing a handler:" (cdr p.v) handler)
                (set-cdr! p.v handler)
                tree)
              (cons (cons (car keys) handler)
                    tree))))))
```

```
#|
;;; Demonstration of handler tree structure.
;;; Note: symbols were used instead of procedures

(define foo (make-generic-operator 3 'foo))
;Value: foo

(pp (get-operator-record foo))
(3)

(defhandler foo 'abc 'a 'b 'c)

(pp (get-operator-record foo))
(3 (a (b (c . abc))))

(defhandler foo 'abd 'a 'b 'd)

(pp (get-operator-record foo))
(3 (a (b (d . abd) (c . abc))))

(defhandler foo 'aec 'a 'e 'c)

(pp (get-operator-record foo))
(3 (a (e (c . aec))
      (b (d . abd)
         (c . abc))))

(defhandler foo 'dbd 'd 'b 'd)

(pp (get-operator-record foo))
(3 (d (b (d . dbd)))
   (a (e (c . aec))
      (b (d . abd)
         (c . abc))))
|#
```