CSE1729	Introduction to Programming Final Exam Solutions	May 6, 2016

1. **(10 points.)** Give short SCHEME functions or short answers for the following.

   (a) **(2 points.)** Explain the difference between *call-by-value* and *call-by-reference*. Explain SCHEME's adoption of these conventions.

   The call-by-value convention passes (a new copy of) the value of an actual parameter to a function—thus any changes made to the formal parameter of the function do not change the calling environment.

   The call-by-reference convention passes a reference to the value of an actual parameter—thus any changes to the value are reflected in the calling environment.

   SCHEME uses call-by-value for all atomic types and call-by-reference for all compound type (pairs, vectors).

   (b) **(2 points.)** Give a SCHEME function `vector-map` which takes two arguments, a vector `v` and a function `f`, and returns the vector of the same length which is obtained by applying `f` to each coordinate of `v`. Thus, once `vector-map` has been correctly defined, one would have the following:

```
> (define (f x) (+ x 1))
> (vector-map (vector 1 2 3) f)
#(2 3 4)
```

```
(define (vector-map v f)
  (let ((result (make-vector (vector-length v))))
    (do ((i 0 (+ i 1)))
      ((= i (vector-length v)) result)
      (vector-set! result i (f (vector-ref v i))))))
```

   (c) **(3 points.)** Give a SCHEME function `duplicates?` which, given a list of numbers, determines if there are any duplicates (that is, if some number appears twice in the list `duplicates?` should return `#t`; otherwise it should return `#f`).

```
(define (duplicates? L)
  (define (element x R)
    (if (null? R) #f
        (or (= x (car R))
            (element x (cdr R)))))
  (if (null? L) #f
      (or (element (car L) (cdr L))
          (duplicates? (cdr L)))))
```

(d) **(3 points.)** Recall the Fibonacci numbers, given by the rule $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ (for integer $n \geq 2$). Give a SCHEME function `fiblist` so that `(fiblist n)` (for $n \geq 1$) returns a list of the Fibonacci numbers

$$(F_n\ F_{n-1}\ \ldots\ F_0)\,.$$

For full credit, your function should be efficient in the sense that it can quickly compute the list of the first 1000 Fibonacci numbers. (Hint: One design criterion that might help you develop an efficient implementation is the following: give an implementation that requires only a single recursive call, and uses no other user-defined functions.)

```
(define (fiblist n)
  (if (= n 1)
      (list 1 0)
      (let ((previous (fiblist (- n 1))))
        (cons (+ (car previous)
                 (cadr previous))
              previous))))
```

2. **(10 points.)** In class, we've talked about several different data structures for representing sets of numbers (e.g., lists, trees, and heaps). Consider an alternate approach to representing sets of numbers using a *function*. The idea is to represent the set $S \subset \{0, 1, \ldots\}$ with a function `f` so that the function `f` takes the value `#t` on those elements of $\{0, 1, \ldots\}$ that are elements of S and the value `#f` on those elements of $\{0, 1, \ldots\}$ that are not in S.

   For example, the set $\{0, 1, \ldots, 100\}$ is represented by the function

```
(define (first-example x)
  (if (and (>= x 0) (<= x 100)) #t #f))
```

   and the set that just contains a particular number, like 237, is represented by the function

```
(define (second-example x)
  (if (= x 237) #t #f))
```

   - if $x \in S$ then `(f x)` evaluates to `#t`, and
   - if $x \notin S$ (and $x \in \{0, 1, \ldots\}$) then `(f x)` evaluates to `#f`.

   This method for representing sets can do some amazing things! For example, it's easy to represent *infinite* sets with this convention: indeed, the set of all even numbers is represented by the function defined as follows:

```
(define (evens x)
  (if (= (modulo x 2) 0) #t #f))
```

Similarly, the set of all numbers greater than 2016 is represented by the function defined as follows:

```
(define (exceeds-this-year x)
  (if (> x 2016) #t #f))
```

Yes, this is going to be awesome!

Now, to determine if an element $x \in \{0, 1, \ldots\}$ is a member of the set represented by `f`, we can use the **member?** function, defined as follows:

```
(define (member? x f)
  (f x))
```

Note that the function **member?** takes two arguments, a number and a set represented by a function, and returns the Boolean value that indicates whether the set contains the number.

Remark. Note that the result of the function on values outside $\{0, 1, \ldots\}$ is irrelevant—if two functions agree on all values in $\{0, 1, \ldots\}$ then we consider them to represent the very same set.

(a) **(3 points.)** Write a function **union** which takes two functions that represent sets (as described previously) and returns the function that represents their *union*. Warning: Your function should take two functions as arguments and return a *function*.

```
(define (union S T)
  (lambda (x) (or (S x) (T x))))
```

(b) **(2 points.)** For a set $S \subset \{0, 1, \ldots\}$, let shift($S$) denote the set of all numbers $x \in \{0, 1, \ldots\}$ so that $x - 1 \in S$. So, shift(S) "shifts the set S over by one." Write a function `shift` so that, given a set, it returns the shift of that set. (Warning: Be careful about the number 0.)

```
(define (shift S)
  (lambda (x)
    (if (= x 0)
        #f
        (S (- x 1)))))
```

(c) **(3 points.)** Write a function `minimum` which, given a *non-empty* subset of $\{0, 1, \ldots\}$ (as a function), returns the minimum element of the set. What happens if your function is given the empty set?

```
(define (minimum S)
  (define (search k)
    (if (S k) k (search (+ k 1))))
  (search 0))
```

This will never terminate if it called with the emptyset.

(d) **(2 points.)** Write a function `factorials` which represents the set $\{n! \mid n \in \{1, \ldots\}\} = \{1, 2, 6, 24, \ldots\}$. (Hint: To figure out if n is in the set, the easiest thing to do is to search over appropriate k to check if $n = k!$.)

```
(define (factorials x)
  (define (factorial n)
    (if (= n 0) 1 (* n (factorial (- n 1)))))
  (define (search k)
    (cond ((< (factorial k) x) (search (+ k 1)))
          ((= (factorial k) x) #t)
          (else #f)))
  (search 0))
```

3. **(10 points.)** Recall that a *heap* is a binary tree which satisfies the *heap condition*: each node of the tree is associated with a numeric value, and the numeric value of any node is less than those of its children. For this problem, use the tree conventions we adopted in class: a node of a tree is represented as a list containing the numeric value of the node and the two subtrees. You may use the convenience functions:

```
(define (make-tree v L R) (list v L R))
(define (value T) (car T))
(define (left T) (cadr T))
(define (right T) (caddr T))
```

(a) **(1 points.)** Write a SCHEME function `heap-min` which, given a heap `H`, returns the smallest value in `H`.

```
(define (heap-min H) (value H))
```

(b) **(3 points.)** Write a SCHEME function `insert` which, given a heap `H` and a number `x`, inserts the value `x` into `H` and returns the resulting heap. Use the *alternating subtree heuristic* for insert to keep the heap balanced: specifically, make sure that subsequent inserts into the same heap will alternate which subtree is used for recursive insertion.

```
(define (insert x H)
  (cond ((null? H) (make-tree x '() '()))
        ((<= x (value H)) (make-tree x
```

```
                                    (right H)
                                    (insert (value H)
                                            (left H))))
        (else (make-tree (value H)
                         (right H)
                         (insert x (left H))))))
```

(c) **(4 points.)** Write a SCHEME function `removemin` which, given a heap `H`, removes the minimum value from the heap and returns the resulting heap.

```
(define (remove-min H)
  (cond ((null? (left H)) (right H))
        ((null? (right H)) (left H))
        ((< (value (left H)) (value (right H)))
         (make-tree (value (left H))
                    (remove-min (left H))
                    (right H)))
        (else (make-tree (value (right H))
                         (left H)
                         (remove-min (right H))))))
```

(d) **(2 points.)** Write a SCHEME function `extract-all` which takes a heap `H` as an argument, and returns a list containing all numbers in the heap in sorted order.

```
(define (extract-all H)
  (if (null? H) '()
      (cons (value H)
            (extract-all (remove-min H)))))
```

4. **(10 points.)** Recall the standard **Set** abstract data type, which defines the operations ***empty?***, ***insert***, and ***member?***. When the set is sure to contain only numbers, you can define the following bonus operation:

   previous Given a value v in the set, return the *previous* value in the set (that is, the largest among all elements in the set that are smaller than v). If ***previous*** is called in an undefined situation—for example, v is the smallest element of the set or the set it empty—you may return a special token `'undefined` which warns the user that there is no "previous" element.

   (a) **(7 points.)** Give a complete implementation of the **Set** abstract data type which also supports the operation ***previous*** above. You should encapsulate everything in a SCHEME object to hide the internals, so your object should have the form:

```
(define (make-set)
  (let ((...))                    ;; internal set variables
    (define (empty?) ...)         ;; set ADT methods
    (define (insert x) ...)
    (define (member? x) ...)
    (define (previous x) ...)
    (define (dispatcher ...) ...) ;;the dispatcher
    dispatcher))
```

```
(define (makeset)
  (let ((elements-inc '())
        (elements-dec '()))
    (define (element? x L)
      (if (null? L) #f
          (or (= x (car L))
              (element? x (cdr L)))))
    (define (insert-sorted-list x L smaller?)
      (cond ((null? L) (list x))
            ((smaller? x (car L)) (cons x L))
            (else (cons (car L)
                        (insert-sorted-list x (cdr L)
                                            smaller?)))))
    (define (member? x) (element? x elements-inc))
    (define (insert x)
      (set! elements-inc
            (insert-sorted-list x elements-inc <))
      (set! elements-dec
            (insert-sorted-list x elements-dec >)))
    (define (find-successor elt L)
      (cond ((null? L) 'undefined)
            ((null? (cdr L)) 'undefined)
            ((= elt (car L)) (cadr L))
            (else (find-successor elt (cdr L)))))
    (define (next x) (find-successor x elements-inc))
    (define (prev x) (find-successor x elements-dec))
    (define (dispatcher method)
      (cond ((eq? method 'empty?)
             (lambda () (null? elements-inc)))
            ((eq? method 'insert) insert)
            ((eq? method 'member?) member?)
            ((eq? method 'prev) prev)
            ((eq? method 'next) next)))
```

```
    dispatcher))
```

(b) **(3 points.)** Your set object above should be able to handle sets containing arbitrary numbers. Now consider an easier case where you may assume that the set only needs to maintain numbers in $\{0, \ldots, N-1\}$ for a specific, known value of N. There is a nice trick for this case: Given the number N, set up a vector with N cells, each of which initially contains the Boolean value `#f`; to indicate that a particular element k has been inserted into the set, set the Boolean value at position k to be `#t`. Now you can handle insertion and membership queries very quickly! However, **`previous`** will continue to be slow, requiring that you step through the vector to find adjacent elements.

Suppose you have a setting where it's acceptable for ***insert*** to be slow, but ***member?*** and ***previous*** must be extremely fast. Explain (in English) how to adapt the vector idea to achieve this. (Hint: Keep a little more information in each cell.)

Use vector, as described above, but in each cell of the vector maintain the previous value (as an integer). Then ***insert*** is a little slow, because you have to search for the previous value, but ***member?*** and ***previous*** are very quick!

5. **(10 points.)** Recall the *Queue* abstract data type, which provides the following basic operations:

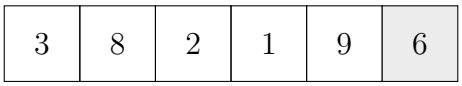
empty? Returns a Boolean value (`#t` or `#f`) depending on whether the queue is empty;

enqueue Adds a new element to the "end" of the queue;

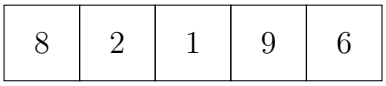
dequeue Removes (and returns) the element at the "front" of the queue.

For simplicity, we just consider queues containing numbers. In this case, you can think of a queue as a row of cells—each containing a number—with a distinguished "front" and "back." Our convention shall be to place the "front" of the queue on the left, and the "back" of the queue on the right. Thus, the diagram

represents a queue where 3 is at the front and 9 is at the back. The enqueue operation adds a new cell to the back of the queue; the dequeue operation removes a cell from the front of the queue. Thus the result of an enqueue operation with the number 6 results in the queue



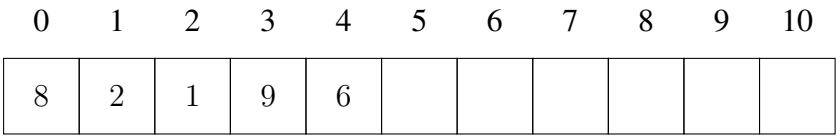
and a subsequent dequeue operation results in the queue



and returns the value 3.

In this problem, you will implement a queue using a *vector*. The idea will be to set up a vector (with a large number of cells) and maintain the queue in an adjacent sequence of cells in the vector.

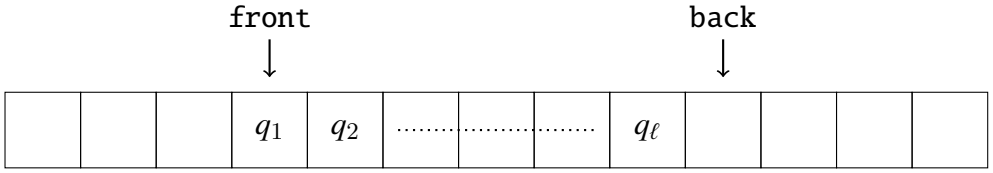
(a) **(2 points.)** Perhaps the most natural approach would be to simply maintain the queue in positions $0, 1, \ldots, \ell - 1$ of the vector (assuming that the queue is currently containing ℓ numbers). Thus, the queue pictured on the previous page would be maintained in the vector by placing the numbers $8, \ldots, 6$ in cells $0, \ldots, 4$ of the vector:



You may assume that `maxsize` is significantly larger than the size of any queue you ever wish to maintain, so that you will never run out of room. Even so, this approach has a serious shortcoming when it comes to the *time required for the basic operations*. What's the problem? (Keep in mind that this convention requires that a queue with ℓ numbers is always held in the first ℓ cells of the vector.)

Note that a dequeue operation will have to be very expensive, as the entire queue must be moved over one cell in memory.

(b) **(6 points.)** Instead of the proposal above, consider the following convention for maintaining the queue: as above, the queue will be maintained in a collection of adjacent cells of the vector; however the positions of the "front" and the "back" may drift over time. More concretely, the implementation will maintain two indices, `front` and `back`, so that the queue occupies cells `front`, . . . , `back` – 1, as in the following diagram:



Concretely,

- When the first element is placed in the queue (via `enqueue`), it is placed at position 0 of the vector and `front` and `back` are set appropriately.
- The enqueue operation places its argument at position `back` and increments `back`.
- The dequeue operation returns the element at position `front` and increments `front`.

Of course, you should encapsulate all your code in a SCHEME object which exposes methods that reflect the abstract datatype operations mentioned above. The function that constructs your queue object should take one parameter: the size of the vector used to represent the queue. Thus, your object should have the form:

```
(define (make-queue maxsize)
  (let ((...))              ;; internal queue variables
    (define (empty?) ...) ;; queue methods
    (define (enqueue x) ...)
    (define (dequeue) ...)
    (define (dispatcher ...) ...) ;;the dispatcher
    dispatcher))
```

Remark. Make sure your object behaves in a reasonable way if `dequeue` is called on an empty queue. As mentioned above, you may pretend that the vector has an infinite number of cells, so there is no risk of `back` ever exceeding `maxsize`.

(c) **(2 points.)** Note that the implementation above can support no more than `maxsize` enqueue operations (because after so many enqueue operations, the variable `back` will hold an index larger than the size of the vector—it will "fall off" the right-hand side of the vector). This is a bit disappointing: even if the queue never grows beyond two elements in size, our implementation breaks down after `maxsize` enqueues. Explain how to fix this problem, so that the implementation can handle *any* sequence of enqueue and dequeue operations, so long as the size of the queue never grows to more than `maxsize` – 1. A brief answer in English will suffice.

Replace increment with "increment mod `maxsize`" so the memory "wraps around." See the implementation below.

```
(define (make-queue maxsize)
  (let ((universe (make-vector maxsize))
        (front 0)
        (back 0))
    (define (empty?) (= front back))
    (define (enqueue x)
      (begin (vector-set! universe back x)
             (set! back (remainder (+ 1 back) maxsize))))
    (define (dequeue)
      (if (empty?) '()
          (let ((result (vector-ref universe front)))
            (begin (set! front
                         (remainder (+ 1 front) maxsize))
                   result))))
    (define (dispatcher method)
      (cond ((eq? method 'empty?) empty?)
            ((eq? method 'enqueue) enqueue)
            ((eq? method 'dequeue) dequeue)))
    dispatcher))
```