Part A.

Let me start with defining the new datatype in data-structures.scm:
Here I am defining a vector as a structure with length and a place where it begins.

```
(vec-val
      (vec vec?))

(define-datatype vec vec?
    (a-vector
     (length integer?)
     (first reference?)))
```

Now we need to make the helper functions for the vector operations we need that will manipulate memory.

```
(define (vec-new length value)
    (if (> length 0)
        (let loop ((i 0) (ref -1))
          (if (= i length)
              (a-vector (- ref (- length 1)) length)
              (loop (+ i 1) (newref value))))
        (eopl:error 'vec-new "Bad Length")))
  (define (vec-zeros length)
    (vec-new length (num-val 0)))


  (define (vec-length vector)
    (cases vec vector
      (a-vector (first length) length)))


  (define (vec-set! vector index value)
    (cases vec vector
      (a-vector (first length)
        (if (and (>= index 0) (< index length))
            (setref! (+ first index) value)
            (eopl:error 'vec-set! "Bad Index")))))


  (define (vec-ref vector index)
    (cases vec vector
      (a-vector (first length)
        (if (and (>= index 0) (< index length))
            (deref (+ first index))
            (eopl:error 'vec-ref "Bad Index")))))


  (define (vec-copy vector)
    (cases vec vector
      (a-vector (first length)
        (let ((copy (vec-zeros length)))
          (let loop ((i 0))
            (if (= i length) copy
                (begin (vec-set! copy i (deref (+ first i))) (loop (+ i 1)))))))))


  (define (vec-swap! vector index1 index2)
    (cases vec vector
      (a-vector (first length)
        (if (and (>= index1 0) (< index1 length) (>= index2 0) (< index2 length))
            (let ((temp (deref (+ first index1))))
              (setref! (+ first index1) (deref (+ first index2)))
              (setref! (+ first index2) temp))
            (eopl:error 'vec-swap! "Bad Index")))))
```

```
(define expval->vec
    (lambda (v)
      (cases expval v
        (vec-val (vec) vec)
        (else (expval-extractor-error 'vec v)))))
```

We also need an extractor to go back and forth between vectors and expvals.

Next, we need to add the functions to the value-of of our interpreter.

```
(newvector-exp (exp1 exp2)
               (let ((length (expval->num (value-of exp1 env)))
                     (value (value-of exp2 env)))
                 (vec-val (vec-new length value))))

(length-vector-exp (exp1)
                   (let ((vec (expval->vec (value-of exp1 env))))
                     (num-val (vec-length vec))))

(update-vector-exp (exp1 exp2 exp3)
                   (let ((vec (expval->vec (value-of exp1 env)))
                         (index (expval->num (value-of exp2 env)))
                         (value (value-of exp3 env)))
                     (vec-set! vec index value)))

(read-vector-exp (exp1 exp2)
                 (let ((vec (expval->vec (value-of exp1 env)))
                       (index (expval->num (value-of exp2 env))))
                   (vec-ref vec index)))

(swap-vector-exp (exp1 exp2 exp3)
                 (let ((vec (expval->vec (value-of exp1 env)))
                       (index1 (expval->num (value-of exp2 env)))
                       (index2 (expval->num (value-of exp3 env))))
                   (vec-swap! vec index1 index2)))

(copy-vector-exp (exp)
                 (let ((vec (expval->vec (value-of exp env))))
                   (vec-val (vec-copy vec))))
```

Now we modify the lang.scm file to add the grammar for new expressions.

```
(expression
 ("newvector" "(" expression "," expression ")")
 newvector-exp)

(expression
 ("update-vector" "(" expression "," expression "," expression ")")
 update-vector-exp)

(expression
 ("read-vector" "(" expression "," expression ")")
 read-vector-exp)

(expression
 ("length-vector" "(" expression ")")
 length-vector-exp)

(expression
 ("swap-vector" "(" expression "," expression "," expression ")")
 swap-vector-exp)

(expression
 ("copy-vector" "(" expression ")")
 copy-vector-exp)
```

Part B.

First we define the queue datatype in the data-structures.scm:

```
(queue-val
      (queue queue?))

(define-datatype queue queue?
    (a-queue
     (entry vec?)
     (start reference?)
     (end reference?)
     (length reference?)))
```

Then we define an extractor:

```
(define expval->queue
    (lambda (v)
      (cases expval v
        (queue-val (queue) queue)
        (else (expval-extractor-error 'queue v)))))
```

Helper functions:

```
(define (queue-new n)
    (a-queue (vec-new n 0)
             (newref 0)
             (newref -1)
             (newref 0)))

  (define (queue-empty? q)
    (cases queue q
      (a-queue (entry start end length)
               (= (deref length) 0))))


  (define (queue-full? q)
    (cases queue q
      (a-queue (entry start end length)
               (= (deref length) (vec-length entry)))))


  (define (set-start! q value)
    (cases queue q
      (a-queue (entry start end length)
               (if (and (>= value 0) (< value (vec-length entry)))
                   (setref! start value)
                   (eopl:error 'set-start! "Bad Index")))))


  (define (set-end! q value)
    (cases queue q
      (a-queue (entry start end length)
               (if (and (>= value 0) (< value (vec-length entry)))
                   (setref! end value)
                   (eopl:error 'set-end! "Bad Index")))))


  (define (queue-enqueue! q value)
    (cases queue q
      (a-queue (entry start end length)
               (if (queue-full? q)
                   (eopl:error 'queue-enqueue! "Full Queue")
                   (begin (set-end! q (modulo (+ (deref end) 1) (vec-length entry)))
                          (vec-set! entry (deref end) value)
                          (setref! length (+ (deref length) 1)))))))


  (define (queue-dequeue! q)
    (cases queue q
      (a-queue (entry start end length)
               (if (queue-empty? q)
                   (num-val -1)
                   (begin (let ((value (vec-ref entry (deref start))))
                            (set-start! q (modulo (+ (deref start) 1) (vec-length entry)))
                            (setref! length (- (deref length) 1))
                            value))))))


  (define (queue-length q)
    (cases queue q
      (a-queue (entry start end length)
               (deref length))))


  (define (queue-peek q)
    (cases queue q
      (a-queue (entry start end length)
               (if (queue-empty? q)
                   (num-val -1)
                   (vec-ref entry (deref start))))))


  (define (queue-print q)
    (cases queue q
      (a-queue (entry start end length)
               (let loop ((i 0) (index (deref start)))
                 (if (= i (deref length))
                     (newline)
                     (begin (display (vec-ref entry i))
                            (display " ")
                     (loop (+ i 1) (modulo (+ index 1) (vec-length entry)))))))))
```

Modify interp:

```
(newqueue-exp (exp)
  (let ((length (expval->num (value-of exp env))))
    (queue-val (queue-new length))))

(enqueue-exp (exp1 exp2)
  (let ((queue (expval->queue (value-of exp1 env)))
        (value (value-of exp2 env)))
    (queue-enqueue! queue value)))

(dequeue-exp (exp)
  (let ((queue (expval->queue (value-of exp env))))
    (queue-dequeue! queue)))

(queue-size-exp (exp)
  (let ((queue (expval->queue (value-of exp env))))
    (num-val (queue-length queue))))

(queue-empty-exp (exp)
  (let ((queue (expval->queue (value-of exp env))))
    (bool-val (queue-empty? queue))))

(peek-queue-exp (exp)
  (let ((queue (expval->queue (value-of exp env))))
    (queue-peek queue)))

(print-queue-exp (exp)
  (let ((queue (expval->queue (value-of exp env))))
    (queue-print queue)))
```

Add grammar:

```
 (expression
 ("newqueue" "(" expression ")")
 newqueue-exp)

(expression
 ("enqueue" "(" expression "," expression ")")
 enqueue-exp)

(expression
 ("dequeue" "(" expression ")")
 dequeue-exp)

(expression
 ("queue-size" "(" expression ")")
 queue-size-exp)

(expression
 ("peek-queue" "(" expression ")")
 peek-queue-exp)

(expression
 ("queue-empty?" "(" expression ")")
 queue-empty-exp)

(expression
 ("print-queue" "(" expression ")")
 print-queue-exp)
```

Part C, Bonus:

First we add the interp expression:

```
        (vec-mult-exp (exp1 exp2)
                      (let ((vec1 (expval->vec (value-of exp1 env)))
                            (vec2 (expval->vec (value-of exp2 env))))
                        (vec-val (vec-mult vec1 vec2))))
```

Then we need to add vec-mult helper to data-structures

```
(define (vec-mult vector1 vector2)
  (cases vec vector1
    (a-vector (head1 length1)
              (cases vec vector2
                (a-vector (head2 length2)
                          (if (= length1 length2)
                              (let ((result (vec-new length1 0)))
                                (let loop ((i 0))
                                  (if (= i length1) result
                                      (begin (vec-set! result i
                                                       (num-mult (deref (+ head1 i)) (deref (+ head2 i))))
                                             (loop (+ i 1))))))
                              (eopl:error 'vec-mult "Bad Length")))))))
```

Finally the lang:

```
      (expression
       ("vec-mult" "(" expression "," expression ")")
       vec-mult-exp)
```