COMP301 Project 5

Ahmet Uyar

This project was done individually. All test cases pass for all parts.

Part A:

First we add the vector data type.

data-structures.rkt: We add the new vector data type. Added its new value type, extractor, and define-datatype expression

```
(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?))
  (proc-val
    (proc proc?))
  (ref-val
    (ref reference?))
  ; ##########################################################
  ; ###### ENTER YOUR CODE HERE
  ; ###### add a new value type for your vectors (and possible for queues)
  ; ##########################################################
  (vec-val ;; adding vec as a new value
   (vec vec?))

  ; ##########################################################
  )


 ;; we add the case for vec here
 (define expval->vec
   (lambda (v)
     (cases expval v
       (vec-val (vec) vec)
       (else (expval-extractor-error 'vec v)))))

;; datatype of vec
(define-datatype vec vec?
  (my-vector
   (first reference?)
   (size integer?)))
```

Then we proceed by adding the grammar for vector operations to **lang.rkt**

```
(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)
```

Adding the corresponding methods to **interp.rkt**

Expression handling: We translate the expressions into their corresponding required value types and pass the raw data into the helper functions where the heavy lifted is occurring

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

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

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

```
        (length-vector-exp (exp1)
                           (let ((my-vector (expval->vec (value-of exp1 env))))
                             (length-vector my-vector)))

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

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

Helper functions: Note that we are using memory operations as stated

```
(define (new-vector size value) ;; new vector
   (if (> size 0)
       (new-vector-loop 0 -1 value size)
       (eopl:error 'new-vector "length of vector should be positive")))

 (define (new-vector-loop i ref value size) ;; helper function
   (if (= i size)
     (my-vector (- ref (- size 1)) size)
     (new-vector-loop (+ i 1) (newref value) value size)))


 (define (update-vector vector index value) ;; update vector
   (cases vec vector
     (my-vector (first size)
                (if (and (> index -1) (> size index))
                    (setref! (+ index first) value)
                    (eopl:error 'update-vector "index out of bounds!")))))

 (define (read-vector vector index) ;; read vector
   (cases vec vector
     (my-vector (first size)
                (if (and (> index -1) (> size index))
                    (deref (+ index first))
                    (eopl:error 'update-vector "index out of bounds!")))))


 (define (length-vector vector) ;; length of vector
   (cases vec vector
     (my-vector (first size)
                (num-val size))))
```

```
  (define (swap-vector vector index1 index2) ;; swap vector
    (cases vec vector
      (my-vector (first size)
                 (if (and (and (> index1 -1) (> size index1)) (and (> index2
-1) (> size index2)))
                     (let ((tmp (deref (+ index1 first))))
                       (setref! (+ index1 first) (deref (+ index2 first)))
                       (setref! (+ index2 first) tmp))
                     (eopl:error 'swap-vector "one of the indices are out of
bounds!")))))

  (define (copy-vector vector) ;; copy vector
    (cases vec vector
      (my-vector (first size)
                 (copy-vector-loop 0 (new-vector size (num-val 0)) first
size))))

  (define (copy-vector-loop i copy first size) ;; helper function
    (if (= i size)
        copy
        (begin
          (update-vector copy i (deref (+ first i)))
          (copy-vector-loop (+ i 1) copy first size))))
```

All test cases for Part A

Part B:

A queue has its front, back, size, and stored data. Adding data type definition, value type, and expression value conversion.

```
(queue-val ;; adding queue
 (queue queue?))

(define-datatype queue queue?
  (my-queue
   (data vec?)
   (front reference?)
   (back reference?)
   (size reference?)))
```

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

Adding the grammar for queue operations in **lang.rkt**:

```
;; Part B

(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)
```

Expression handling: We translate the expressions into their corresponding required value types and pass the raw data into the helper functions where the heavy lifted is occurring

```
;; Part B

(newqueue-exp (exp1)
              (let ((max-size (expval->num (value-of exp1 env))))
                (queue-val (new-queue max-size))))

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

(dequeue-exp (exp1)
              (let ((queue (expval->queue (value-of exp1 env))))
                (dequeue-queue queue)))

(queue-size-exp (exp1)
                (let ((queue (expval->queue (value-of exp1 env))))
                  (num-val (size-queue queue))))

(peek-queue-exp (exp1)
                (let ((queue (expval->queue (value-of exp1 env))))
                  (peek-queue queue)))

(queue-empty-exp (exp1)
                 (let ((queue (expval->queue (value-of exp1 env))))
                   (bool-val (empty-queue? queue))))

(print-queue-exp (exp1)
                 (let ((queue (expval->queue (value-of exp1 env))))
                   (print-queue queue)))
```

Helper functions:

New-queue: creates new queue
Enqueue: adds value to the queue
Dequeue: pops front
Size-queue: return size
Peek-queue: reads peek
Empty-queue, full-queue: checker for emptiness and fullness
Print-queue: prints the queue.

```
(define (new-queue l)
  (my-queue (new-vector l 0)
            (newref 0)
            (newref -1)
            (newref 0)))

(define (enqueue-queue que value)
(cases queue que
```

```
    (my-queue (data front back size)
              (if (full-queue? que)
                  (eopl:error 'enqueue-queue "queue is full, cannot insert new
element")
                  (begin
                    (setref! back (modulo (+ 1 (deref back)) (length-vector
data)))
                    (update-vector data (deref back) value)
                    (setref! size (+ 1 (deref size)))
                    que)))))

  (define (dequeue-queue que)
  (cases queue que
    (my-queue (data front back size)
              (if (empty-queue? que)
                  -1
                  (let ((value (read-vector data (deref front))))
                    (setref! front (modulo (+ 1 (deref front)) (length-vector
data)))
                    (setref! size (- (deref size) 1))
                    value)))))

  (define (size-queue que)
    (cases queue que
      (my-queue (data front back size)
                (num-val (deref size)))))

  (define (peek-queue que)
  (cases queue que
    (my-queue (data front back size)
      (if (empty-queue? que)
          (eopl:error 'peek-queue "queue is empty, no element to peek.")
          (read-vector data (deref front))))))

  (define (empty-queue? que)
    (cases queue que
      (my-queue (data front back size)
                (= (deref size) 0))))

  (define (full-queue? que) ;; helper function for full check.
    (cases queue que
      (my-queue (data front back size)
                 (= (deref size) (length-vector data)))))

  (define (print-queue que)
    (cases queue que
      (my-queue (data front back size)
```

```
                (print-queue-helper data (deref front) (deref size)
(length-vector data)))))

  (define (print-queue-helper data index count length) ;; helper for print-que
    (if (= count 0)
        (begin
          (newline))
        (begin
          (display (read-vector data index))
          (display " ")
          (print-queue-helper data (modulo (+ index 1) length) (- count 1)
length))))
```

All test cases pass for Part B

Part C - Bonus:

Adding vec-mult to the grammar:

```
;; Part C

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

Adding its expression handler, passing raw vectors inside vec-val helper function

```
;; Part C

(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))))
```

Helper functions: vec-mult also has a helper function which takes care of the recursion. Vec-mult is mostly for obtaining the values from the two vectors.

```
(define (vec-mult vector1 vector2)
   (cases vec vector1
      (my-vector (first1 size1)
                 (cases vec vector2
                   (my-vector (first2 size2)
                              (if (= size1 size2)
                (let ((result (new-vector size1 0)))
                   (vec-mult-helper first1 first2 result 0 size1)
```

```
                                    result)
                                   (eopl:error 'vec-mult "Vectors
dimension mismatch.")))))))

  (define (vec-mult-helper first1 first2 result index size) ;; helper
;; function for filling the vector
    (if (< index size)
        (begin
          (let ((val1 (expval->num (deref (+ first1 index))))
                (val2 (expval->num (deref (+ first2 index)))))
            (update-vector result index (num-val (* val1 val2))) ;;
update the result's index
            (vec-mult-helper first1 first2 result (+ index 1) size)))
;; recursion
        result))
```

All tests pass for Part C