COMP301 Project 5

Project completed by Altun Hasanli. Code passess all tests provided.

Part A.

Vector Datatype

Define the new type for vectors and the extractor for it.

- `head` is the first element in `the-store`.
- `length` is the fixed capacity of the vector.

```
(define-datatype expval expval?
  ...
  (vec-val
      (vec vec?))
  ...)

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

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

Vector Operations

Define methods to operate on the new `vec` type.

- `vec-new` allocates a fresh vector with a fixed length and a contingent list of values initialized to `value`.
- `vec-copy` allocates a new vector of all zeros, and copies elements to the corresponding locations.
- `vec-set!`, `vec-ref`, etc. dereference and change the contents of references in a similar way to pointer arithmetic, adding and subtracting index positions from the `head` reference. e.g. `(deref (+ head index))`.

```
; vec-new : Int x ExpVal -> Vec
(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 "Length must be greater than 0")))

; vec-zeros : Int -> Vec
(define (vec-zeros length)
  (vec-new length (num-val 0)))

; vec-set! : Vec x Int x ExpVal -> Void
(define (vec-set! vector index value)
  (cases vec vector
    (a-vector (head length)
      (if (and (>= index 0) (< index length))
          (setref! (+ head index) value)
          (eopl:error 'vec-set! "Index out of bounds")))))

; vec-length : Vec -> Int
(define (vec-length vector)
  (cases vec vector
    (a-vector (head length) length)))

; vec-ref : Vec x Int -> ExpVal
(define (vec-ref vector index)
  (cases vec vector
    (a-vector (head length)
      (if (and (>= index 0) (< index length))
          (deref (+ head index))
          (eopl:error 'vec-ref "Index out of bounds")))))

; vec-copy : Vec -> Vec
(define (vec-copy vector)
  (cases vec vector
    (a-vector (head length)
      (let ((copy (vec-zeros length)))
        (let loop ((i 0))
          (if (= i length) copy
              (begin (vec-set! copy i (deref (+ head i))) (loop (+ i 1))))))))))

; vec-swap! : Vec x Int x Int -> Void
(define (vec-swap! vector index1 index2)
  (cases vec vector
    (a-vector (head length)
      (if (and (>= index1 0) (< index1 length) (>= index2 0) (< index2 length))
          (let ((temp (deref (+ head index1))))
            (setref! (+ head index1) (deref (+ head index2)))
            (setref! (+ head index2) temp))
          (eopl:error 'vec-swap! "Index out of bounds")))))
```

Extend the grammar for the new vector operations.

```
...
(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)
...
```

Extend the interpreter to handle vector operations.

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

(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)))

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

(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))))
...
```

Part B.

Queue Datatype

Define the new type for queues and the extractor for it.

- `data` is a vector that stores the data.
- `front` is a reference to the front element in the vector.
- `rear` is a reference to the rear element in the vector.
- `size` is the number of elements in the vector.

```
(define-datatype expval expval?
        ...
        (queue-val
      (queue queue?))
 ...)

(define-datatype queue queue?
    (a-queue
     (data vec?)
     (front reference?)
     (rear reference?)
     (size reference?)))

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

Queue Operations

Define methods to operate on the `queue` type.

- Queue is implemented as a circular buffer using the underlying vector.
- `queue-new` allocates `n + 3` elements in the store, initializes the
- `enqueue` and `dequeue` use memory operations to modify the front and rear references in `the-store` and to increment/decrement the `size`.
- Data in the vector is modified using the vector operations `vec-ref`, `vec-set!`, etc.
- When the queue is full or the provided indices are out of bounds, corresponding errors are thrown.
- When the queue is empty, peek and dequeue operations both return `-1`.

```
; queue-new : Int -> Queue
(define (queue-new n)
  (a-queue (vec-new n 0)
           (newref 0)
           (newref -1)
           (newref 0)))

; queue-empty? : Queue -> Bool
(define (queue-empty? q)
  (cases queue q
    (a-queue (data front rear size)
      (= (deref size) 0))))

; queue-full? : Queue -> Bool
(define (queue-full? q)
  (cases queue q
    (a-queue (data front rear size)
      (= (deref size) (vec-length data)))))

; set-front! : Queue x Int -> Void
(define (set-front! q value)
  (cases queue q
    (a-queue (data front rear size)
      (if (and (>= value 0) (< value (vec-length data)))
          (setref! front value)
          (eopl:error 'set-front! "Index out of bounds")))))

; set-rear! : Queue x Int -> Void
(define (set-rear! q value)
  (cases queue q
    (a-queue (data front rear size)
      (if (and (>= value 0) (< value (vec-length data)))
          (setref! rear value)
          (eopl:error 'set-rear! "Index out of bounds")))))

; queue-enqueue! : Queue x ExpVal -> Void
(define (queue-enqueue! q value)
  (cases queue q
    (a-queue (data front rear size)
      (if (queue-full? q)
          (eopl:error 'queue-enqueue! "Queue is full")
          (begin (set-rear! q (modulo (+ (deref rear) 1) (vec-length data)))
                 (vec-set! data (deref rear) value)
                 (setref! size (+ (deref size) 1)))))))

; queue-dequeue! : Queue -> ExpVal
(define (queue-dequeue! q)
  (cases queue q
    (a-queue (data front rear size)
      (if (queue-empty? q)
          (num-val -1)
          (begin (let ((value (vec-ref data (deref front))))
                   (set-front! q (modulo (+ (deref front) 1) (vec-length data)))
                   (setref! size (- (deref size) 1))
                   value))))))

; queue-size : Queue -> Int
(define (queue-size q)
  (cases queue q
    (a-queue (data front rear size)
      (deref size))))

; queue-peek : Queue -> ExpVal
(define (queue-peek q)
  (cases queue q
    (a-queue (data front rear size)
      (if (queue-empty? q)
          (num-val -1)
          (vec-ref data (deref front))))))

; queue-print : Queue -> Void
(define (queue-print q)
  (cases queue q
    (a-queue (data front rear size)
      (let loop ((i 0) (index (deref front)))
        (if (= i (deref size))
            (newline)
            (begin (display (vec-ref data i))
                   (display " ")
                   (loop (+ i 1) (modulo (+ index 1) (vec-length data)))))))))
```

Extending the language

Extend the grammar for the new queue operations.

```
...
(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)
...
```

Extend the interpreter to handle the new queue operations.

```
...
(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-size 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)))
...
```

Part C.

Vector Multiplication Operation

Define multiplication method for the vector type:

- Allocates a new vector for the multiplication result.
- Iterates over both vectors using `deref` and sets the corresponding element in the resulting vector using `vec-set!` (that uses `set-ref!` in its implementation).
- Returns the resulting vector.

```
; vec-mult : Vec x Vec -> Vec
(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 "Vectors must be of same length")))))))

; num-mult : NumVal x NumVal -> NumVal
(define (num-mult num1 num2)
  (let ((n1 (expval->num num1))
        (n2 (expval->num num2)))
    (num-val (* n1 n2))))
```

Extending the language

Extend the grammar and the interpreter to support vector multiplication.

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

```
...
(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))))
...
```