Project 5

COMP301 Spring 2023

AYDA KANIL - 0064641
YAKUP ENES GÜVEN - 0064045

1. WORKLOAD

1. Ayda Kanıl
   - Part - B Coding
   - Part - A Coding (help)
   - Report

- Yakup Enes Güven (coding mostly done by me)
  - Part - B Coding (help) (partly working)
  - Part - A Coding (working)
  - Part - C Coding (working)
  - Report

2. PART - A

In this part we add vector to EREF. In Figure 1 we implement new operators newvector, update-vector, read-vector, length-vector, and swap-vector with the given definitions in **lang.scm**.

```
(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)
```

Figure 1

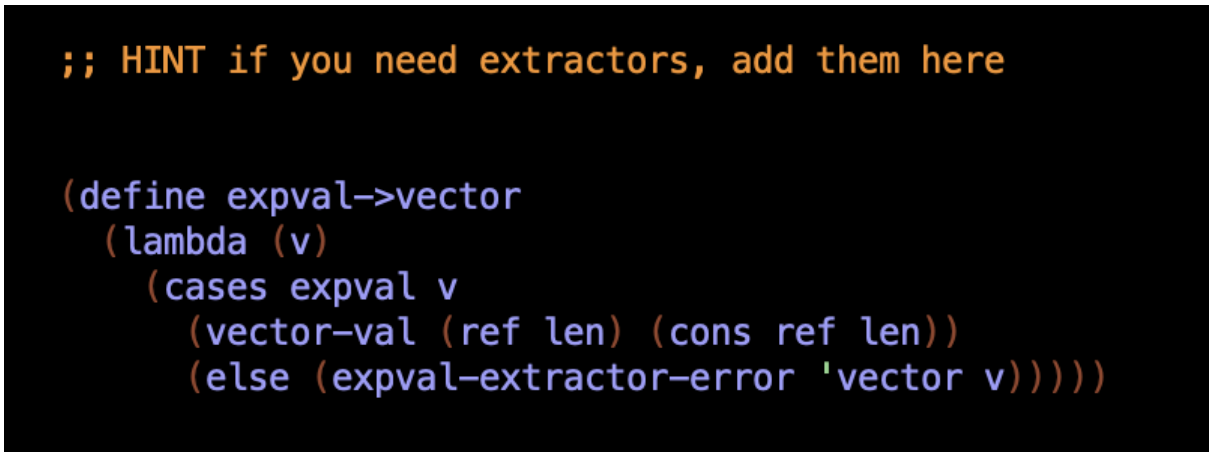
Then, we added vector-val and pair-val to expval to **data-structures.scm** class as it's shown in Figure 2.

```
; ###################################################
; ###### ENTER YOUR CODE HERE
; ###### add a new value type for your vectors (and possible for queues)
; ###################################################
(vector-val
 (ref reference?)
 (len number?)
 )
(pair-val
 (pair pair?))

; ###################################################
)
```

Figure 2

In Figure 3 we added the extractors.



```
;; HINT if you need extractors, add them here


(define expval->vector
  (lambda (v)
    (cases expval v
      (vector-val (ref len) (cons ref len))
      (else (expval-extractor-error 'vector v)))))
```

Figure 3

After these, we first define the vector. Then, we have included the following functions, in **data_structures.scm**, to be used in the interp.scm file.

```
(define-datatype vector vector?
  [an-vector [ref reference?]
             [length (lambda (x)
                       (and (integer? x)
                            (positive? x)))]])



(define make-vector
  (lambda (length value)
    (letrec ((save-to-store
              (lambda (length)
                (if (= length 0)
                    '()
                    (let ((new (newref value)))
                      (save-to-store (- length 1)))))))
      (let ((first (newref value)))(save-to-store (- length 1))first))))

(define vector-ref
  (lambda (vector index)


    (begin

      (deref (+ vector index)))

    ))

(define set-vector!
  (lambda (vector index val)
    (setref! (+ vector index) val)))

(define length-vector
  (lambda(vector)
    (let ((expval (deref vector)))
      (if (null? expval) (length-vector (+ vector 1))
          (let ((val (expval->num expval)))
            (if (= val -1) 0 (+ 1 (length-vector (+ vector 1)))))))))
```

```
(define copy-vector
  (lambda (vec1 num)
    (let ((resvec (make-vector (+ 1 num) 0)))
      (letrec((vec-copy-helper
               (lambda (v1  n )
                 (if (> n 0 )
                     (begin
                       (set-vector! resvec n (num-val (expval->num  (vector-ref v1 n))))
                       (vec-copy-helper  v1   (- n 1) )
                       )
                     (set-vector! resvec n (num-val (expval->num  (vector-ref v1 n) )))
                     ))))

        (vec-copy-helper vec1  num)

        (vector-val resvec (+ 1 num))

        ))))
```

Figure 4

In interp.scm class, we implemented the vector expressions as shown in Figure 5.

```
;; VECTOR-EXP
; newvector(length, value): initializes a vector of size length with the value value.
(newvector-exp (length-exp val-exp)
               (let ((len (expval->num (value-of length-exp env)))
                     (val (value-of val-exp env)))
                 (vector-val (make-vector len val) len)))

; update-vector(vec, index, value): updates the value of the vector vec at index index by value value.
(update-vector-exp (exp1 exp2 exp3)
                   (let ((v1 (value-of exp1 env))
                         (v2 (value-of exp2 env))
                         (v3 (value-of exp3 env)))
                     (let ((vec (car (expval->vector v1)))
                           (index (expval->num v2)))
                       (set-vector! vec index v3))))

; read-vector(vec, index): returns the element of the vector vec at index index.
(read-vector-exp (exp1 exp2)
                 (let ((v1 (value-of exp1 env))
                       (v2 (value-of exp2 env)))
                   (let ((vec (car (expval->vector v1)))
                         (index (expval->num v2)))
                     (vector-ref vec index))))

; length-vector(vec) returns the length of the vector vec.
(length-vector-exp (exp1)
                   (let ((v1 (value-of exp1 env)))
                     (let ((vec (cdr (expval->vector v1))))
                       (num-val vec))))

; swap-vector(vec, index, index): swaps the values of the indexes in the vector vec.
(swap-vector-exp (exp1 exp2 exp3)
                 (let ((v1 (value-of exp1 env))
                       (v2 (value-of exp2 env))
                       (v3 (value-of exp3 env)))
                   (let ((vec (car (expval->vector v1)))
                         (index1 (expval->num v2))
                         (index2 (expval->num v3)))
                     (let ((temp (vector-ref vec index1)))
                       (set-vector! vec index1 (vector-ref vec index2))
                       (set-vector! vec index2 temp)))))


; copy-vector(vec): initializes an new vector with the same values of the given vector vec.
(copy-vector-exp (exp)
                 (let ((vec (car (expval->vector (value-of exp env)))))
                   (copy-vector vec (- (cdr (expval->vector (value-of exp env))) 1))  ))
```

Figure 5

3. PART - B

In this part, we implemented a Queue using vectors that we implemented in Part A.

First, we add the language in lang.scm as shown in Figure 6.

```
(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)
```

Figure 6

Then, in interp.scm class we use vectors to implement a Queue as shown in Figure 7.

```
;; QUEUE-EXP
; newqueue(L) returns an empty queue with max-size L.
(newqueue-exp (L)
              (let ((max-size (value-of L env)))
                (let ((queue (make-vector max-size '())))
                  (vector-val queue))))



; enqueue(q, val) adds the element val to the queue q. If the queue is full it throws a stack overflow error.
(enqueue-exp (queue-exp val-exp)
             (let ((v1 (value-of queue-exp env))
                   (val (value-of val-exp env)))
               (let ((size (vector-ref (expval->vector v1) 0))
                     (elements (vector-ref (expval->vector v1) 1)))
                 (if (= (length elements) size)
                     (eopl:error "Queue overflow")
                     (begin
                       (set-vector! (expval->vector v1) 1 (append elements (list val)))
                       (expval->num (length elements)))))))

; dequeue(q) removes the first element of the queue q and returns its value.
(dequeue-exp (queue-exp)
             (let ((v1 (value-of queue-exp env)))
               (let ((elements (vector-ref (expval->vector v1) 1)))
                 (if (null? elements)
                     (num-val -1)
                     (let ((first-element (car elements)))
                       (begin
                         (set-vector! (expval->vector v1) 1 (cdr elements))
                         first-element)))))))

; queue-size(q) returns the number of elements in the queue q.
(queue-size-exp (queue-exp)
                (let ((v1 (value-of queue-exp env)))
                  (length (vector-ref (expval->vector v1) 1)))))

; peek-queue(q) returns the value of the first element in the queue q without removal.
(peek-queue-exp (queue-exp)
                (let ((v1 (value-of queue-exp env)))
                  (let ((elements (vector-ref (expval->vector v1) 1)))
                    (if (null? elements)
                        (num-val -1)
                        (car elements)))))

; queue-empty?(q) returns true if there is no element inside the queue q and false otherwise.
(queue-empty-exp (queue-exp)
                 (let ((v1 (value-of queue-exp env)))
                   (let ((elements (vector-ref (expval->vector v1) 1)))
                     (null? elements)))))

; print-queue(q) prints the elements in the queue q.
(print-queue-exp (queue-exp)
                 (let ((v1 (value-of queue-exp env)))
                   (let ((elements (vector-ref (expval->vector v1) 1)))
                     (for-each (lambda (element)
                                 (display element)
                                 (display " "))
                               elements)
                     (newline)))))
```

Figure 7

Also, we implemented some helper functions as shown in Figure 8.

```
; ###### YOU CAN WRITE HELPER FUNCTIONS HERE
(define read-vector
  (lambda (vector length)
    (let ((expval (vector-ref vector length)))
      (if (null? expval) (read-vector vector (+ length 1))
          (let ((val (expval->num expval)))
            (if (= val -1) '() (cons val (read-vector vector (+ length 1)))))))))

(define print-vector
  (lambda (lst)
    (if (null? lst) '()
        (letrec ((print-lst
                  (lambda (lst)
                    (display (car lst))
                    (display " ")
                    (if (null? (cdr lst))
                        (display "")
                        (print-lst (cdr lst))))))
          (print-lst lst)))))
```

Figure 8

4. PART - C

In this part, we implemented the language for vec-mult-exp in **lang.scm** class as shown in Figure 9.

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

Figure 9

We implemented the definition of vec-mult function in **data_structures.scm** class as shown in Figure 10.

```
(define vec-mult

  (lambda (vec1 vec2 num)
    (let ((resvec (make-vector (+ 1 num) 0)))
      (letrec((vec-mult-helper
               (lambda (v1 v2 n )
                 (if (> n 0 )
                     (begin
                       (set-vector! resvec n (num-val (* (expval->num  (vector-ref v1 n)) (expval->num  (vector-ref v2 n)))))
                       (vec-mult-helper  v1  v2 (- n 1) )
                       )
                     (set-vector! resvec n (num-val (* (expval->num  (vector-ref v1 n)) (expval->num  (vector-ref v2 n)))))
                     ))))

        (vec-mult-helper vec1 vec2 num)

        (vector-val resvec (+ 1 num))

        ))))
```

Figure 10

Then, we implemented the vec-mult in **interp.scm** class as shown in Figure 11.

```
; vec-mult-exp (vec1, vec2): takes two vectors, calculates their pairwise multiplication and outputs a new vector. If the sizes of the vectors are not equal, it throws an error
(vec-mult-exp (exp1 exp2)
              (let ((vec1 (expval->vector (value-of exp1 env)))
                    (vec2 (expval->vector (value-of exp2 env))))
                (let ((result (vec-mult (car vec1) (car vec2) (- (cdr vec1) 1))))
                  (if (not (eq? (cdr vec1) (cdr vec2)))
                      (eopl:error  "vectors must have the same length") result))))
```

Figure 11