CS61A Notes 11 – Concurrent Vectors In The Sky [Solutions v1.0]

Just When You Were Getting Used To Lists…

QUESTIONS

1. **Write procedure `(sum-of-vector v)` that adds up the numbers inside the vector.**

   You can use an accumulator or a helper like this:

```
(define (sum-of-vector v)
   (define (helper index)
      (cond ((= index (vector-length v)) 0)
            (else (+ (vector-ref v index) (helper (+ index 1))))))
   (helper 0))
```

2. **Write procedure `(vector-copy! src src-start dst dst-start length)`. After the call, `length` elements in vector `src` starting from index `src-start` should be copied into vector `dst` starting from index `dst-start`.**

```
STk> a ==> #(1 2 3 4 5 6 7 8 9 10)
STk> b ==> #(a b c d e f g h i j k)
STk> (vector-copy! a b 5 2 3) ==> okay
STk> a ==> #(1 2 3 4 5 c d e 9 10)
STk> b ==> #(a b c d e f g h i j k)
(define (vector-copy! src src-start dst dst-start length)
   (if (> length 0)
       (begin (vector-set! dst dst-start (vector-ref src src-start))
              (vector-copy! src (+ src-start 1) dst
                     (+ dst-start 1) (- length 1)))))
```

3. **Write procedure `(insert-at! v i val)`; after a call, vector `v` should have `val` inserted into location `i`. All elements starting from location `i` should be shifted down. The last element of `v` is disgarded.**

```
STk> a ==> #(cs61a is cool #[unbound] #[unbound])
STk> (insert-at! a 2 'very) ==> okay
STk> a ==> #(cs61a is very cool #[unbound])
(define (insert-at! v i val)
   (let* ((amt-to-shift (- (vector-length v) i 1))
          (temp-v (make-vector amt-to-shift)))
      (vector-copy! v i temp-v 0 amt-to-shift)
      (vector-copy! temp-v 0 v (+ i 1) amt-to-shift)
      (vector-set! v i val)))
```

   NOTE: why didn't we just do (vector-copy! v i v (+ i 1) amt-to-shift)?

4. **Write procedure `(vector-double! v)`. After a call, vector `v` should be doubled in size, with all the elements in the old vector replicated in the second half. So,**

```
STk> a ==> #(1 2 3 4)
STk> (vector-double! a) ==> okay
STk> a ==> #(1 2 3 4 1 2 3 4)
```

   IMPOSSIBLE! Hope you weren't fooled by this. To double the size of a vector, you'd have to allocate a new vector. However, recall that you cannot change what "a" points to from within a procedure! You can at most *return* a new vector double in size.

5. **Write procedure `(reverse-vector! v)` that reverses the elements of a vector, obviously.**

```
;; for elements from start to stop, do body
(define (from-to-do start stop body)
   (if (> start stop) 'done
       (begin (body start) (from-to-do (+ 1 start) stop body)) ))


(define (reverse-vector! v)
   (from-to-do 0 (- (quotient (vector-length v) 2) 1)
      (lambda (i)
         (let ((temp (vector-ref v i)))
            (vector-set! v i (vector-ref v (- (vector-length v) i 1)))
            (vector-set! v (- (vector-length v) i 1) temp)))))
```

6. **Write procedure `(square-table! t)` that takes in an `n`-by-`m` table and squares every element.**

```
(define (square-table! t)
   (let ((n (vector-length t))
         (m (vector-length (vector-ref t 0))))
      (from-to-do 0 (- n 1)
         (lambda(i)
            (from-to-do 0 (- m 1)
               (lambda(j)
                  (vector-set!
                     (vector-ref! t i)
                     j
                     (square (vector-ref! (vector-ref! t i) j)))))))))
```

A Concurrent March Through Programming Hell

QUESTION: What are the possible values of x after the below?

```
(define x 5)
(parallel-execute (lambda() (set! x (* x 2)))
                  (lambda() (if (even? x) (set! x (+ x 1))
                                          (set! x (+ x 100)))))
```

11, 210, 10, 105, 110

Concurrency: The Series

1. **Here is an attempt to simulate this behavior:**

```
(define (eat-talk i)
   (define (loop)
      (cond ((can-eat? i)
             (take-chopsticks i)
             (eat-a-while)
             (release-chopsticks i))
            (else (spew-nonsense)))
      (loop)
   (loop))

(parallel-execute (lambda() (eat-talk 0))
                  (lambda() (eat-talk 1))
                  (lambda() (eat-talk 2)))
```

```
;; a list of chopstick status, #t if usable, #f if taken
(define chopsticks '(#t #t #t))

;; does person i have both chopsticks?
(define (can-eat? i)
   (and (list-ref chopsticks (right-chopstick i))
        (list-ref chopsticks (left-chopstick i))))

;; let person i take both chopsticks
;; assume (list-set! ls i val) destructively sets the ith element of
;; ls to val
(define (take-chopsticks i)
   (list-set! chopsticks (right-chopstick i) #f)
   (list-set! chopsticks (left-chopstick i) #f))

;; let person i release both chopsticks
(define (release-chopsticks i)
   (list-set! chopsticks (right-chopstick i) #t)
   (list-set! chopsticks (left-chopstick i) #t))

;; some helper procedures
(define (left-chopstick i) (if (= i 2) 0 (+ i 1)))
(define (right-chopstick i) i)
```

Is this correct? What kind of hazard does this create?

Incorrect; more than one person could be eating at once (all three check they can eat, all three take chopsticks, and all three eat)

2. **Here's a proposed fix:**
```
(define protector (make-serializer))
(parallel-execute (protector (lambda() (eat-talk 0)))
                  (protector (lambda() (eat-talk 1)))
                  (protector (lambda() (eat-talk 2)))
```
Does this work?

Unfair. Note that eat-talk generates in infinite loop. The serializer makes sure only one of the three is executed at once, so once parallel-execute picks one to execute, it's going to keep eating and eating, and the others won't even get to execute at all.

3. **Here's another proposed fix: use one mutex per chopstick, and acquire both before doing anything:**
```
(define protectors
   (list (make-mutex) (make-mutex) (make-mutex)))

(define (eat-talk i)
   (define (loop)
      ((list-ref protectors (right-chopstick i)) 'acquire)
      ((list-ref protectors (left-chopstick i)) 'acquire)
      (cond ... ;; as before)
      ((list-ref protectors (right-chopstick i)) 'release)
      ((list-ref protectors (left-chopstick i)) 'release)
      (loop))
   (loop))
```
Does that work?

Deadlock. Suppose all three grab the chopstick on the left at the same time; then all three will be waiting for the chopstick on the right, resulting in deadlock.

4. **What about this:**

```
(define m (make-mutex))
(define (eat-talk i)
   (define (loop)
      (m 'acquire)
      (cond ... ;; as before)
      (m 'release)
      (loop))
   (loop))
```
Inefficient (and not very correct). Only one will eat at the same time, and all other politicians will just be waiting to acquire the mutex (rather than spewing nonsense)

5. **So what would be a good solution?**

```
(define m (make-mutex))
(define (eat-talk i)
   (define (loop)
      (m 'acquire)
      (cond ((can-eat? i)
             (take-chopsticks i)
             (m 'release)
             (eat-a-while)
             (m 'acquire)
             (release-chopsticks i)
             (m 'release))
            (else (m 'release) (spew-nonsense)))
      (loop)
   (loop))
```

Note what we're using the mutex to protect – the chopsticks list structure! Every time we want to look at it or change it, we must be holding the mutex. It's correct because no two processes will be modifying the list at the same time. It's efficient because when we do things that take a long time – like eating or spewing nonsense – we're not holding the mutex.