Concurrency
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

The Dining Politicians Problem
Politicians like to congregate once in a while, eat and spew nonsense. For fun, one slow Saturday afternoon, three politicians meet to have such wild fun. They sit around a circular table; however, due to the federal deficit, they are provided with only three chopsticks, each lying in between two people. A politician will be able to eat only when both chopsticks next to him are not being used. If he cannot eat, he will just spew nonsense.

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?

Way wrong. 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. Two people won’t get to eat.

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?

Nope. Deadlock. All three people grab the chopsticks on their right and wait for the left chopstick.

4. What about this:

(define m (make-mutex))
(define (eat-talk i)
 (define (loop)
   (m 'acquire)
   (cond ... ;; as before)
   (m 'release)
   (loop))
 (loop))

Nope, politicians don’t spew nonsense while not eating. Inefficient.
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))