Exercise 2.22.

Louis Reasoner tries to rewrite the firstsquare-listprocedure of exercise 2.21 so that it evolves an iterative process:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

Unfortunately, defining `square-list` this way produces the answer list in the reverse order of the one desired. Why?

Louis then tries to fix his bug by interchanging the arguments to `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

This doesn't work either. Explain.

Answer.

We can reveal the reason why Louis's procedure obtains the answer list in a reverse order by traceing the process it generated using substitution model while evaluating an expression, say, `(square-list (list 1 2 3 4))`:

```
(square-list (list 1 2 3 4))
(iter (list 1 2 3 4) nil)
;(iter (list 2 3 4) (cons (square 1) nil))
(iter (list 2 3 4) (list 1))
;(iter (list 3 4) (cons (square 2) (list 1)))
(iter (list 3 4) (list 4 1))
;(iter (list 4) (cons (square 3) (list 4 1)))
(iter (list 4) (list 9 4 1))
;(iter nil (cons (square 4) (list 9 4 1)))
(iter nil (list 16 9 4 1))
(list 16 9 4 1)
(16 9 4 1)
```

The processes generated above indicates that Louis's procedure "`cons`es up" an answer list while `car`ing down a list. This obviously give rise to the scenario Louis encounters. In fact, if we eliminate `square` in the body of `iter`, this procedure immediately turns into the procedure `reverse` in exercise 2.18 which generates iterative process.

Again, we can investigate the behavior of Louis's "fixed version" of his `square-list` procedure by evaluating `(square-list (list 1 2 3 4))`:

```
(square-list (list 1 2 3 4))
;(iter (list 2 3 4) (cons nil (square 1)))
(iter (list 2 3 4) (list 1))
;(iter (list 3 4) (cons (list 1) (square 2)))
(iter (list 3 4) (list (list 1) 4))
```

```
;(iter (list 4) (cons (list (list 1) 4) (square 3)))
(iter (list 4) (list (list (list 1) 4) 9))
;(iter nil (cons (list (list (list 1) 4) 9) (square 4)))
(iter nil (list (list (list (list 1) 4) 9) 16))
(list (list (list (list 1) 4) 9) 16)
(((( . 1) . 4) . 9) . 16)
```

Things even go worse, for what produced by this new `square-list` procedure is a nested list, far away from what one desired. We see that the new `square-list` procedure `cons`es the `answer`, which is a list, onto the `square-list` of subsequent elements.

We see that the answer list generated by Louis's first `square-list` procedure is in a reverse order of one desired. Thus, we can fix it by the law of contraries—make the `square-list` pass a reverse list of the original to its internal procedure `iter` while calling to it:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter (reverse items) nil))
```