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 extract the reason why Louis's procedure obtained 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)
```

This process indicates that Louis's procedure "conses up" an answer list while caring down a list. Therefore, what obtained from Louis's procedure is a answer list in the reverse order of the one desired. In fact, if we eliminate `square` in the body of `iter`, this procedure immediately becomes the procedure `reverse` in exercise 2.18 which generates iterative process.

Again, we can investigate the performance of Louis's "fixed version" of this 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)
```

This even makes the thing worse, for what produced by this procedure is a nested list which one does not desire. We see that it `cons`es the `answer` which is a list onto the squared value of each subsequent element in the original list. Hence, a nested list came into being.

As we've seen, the answer list generated by Louis's first procedure is in a reverse order of one desired. Thus, we can fix it right by the law of contraries, just pass a reverse list of the original to the internal procedure `iter` while invoking:

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