Exercise 4.27.

Suppose we type in the following definitions to the lazy evaluator:

```
(define count 0)

(define (id x)
  (set! count (+ count 1))
  x)
```

Give the missing values in the following sequence of interactions, and explain your answers.[1]

```
(define w (id (id 10)))

;;; L-Eval input:
count
;;; L-Eval value:
<response>

;;; L-Eval input:
w
;;; L-Eval value:
<response>

;;; L-Eval input:
count
;;; L-Eval value:
<response>
```

Answer.

Starting the the evaluator we can see the response it prompted:

```
(define w (id (id 10)))

;;; L-Eval input:
count
;;; L-Eval value:
1
```

We know that the lazy evaluator will not evaluate the arguments until the body of a procedure is entered. As you can see, `(id (id 10))` is a procedure application. By lazy evaluation, the evaluator packaged the argument `(id 10)` to produce a thunk and evaluated the body of the compound procedure `id`, that is,

```
((set! count (+ count))
 x)
```

Evaluating this increments the variable `count` to 1 and unchangedly return the argument—the thunk it just packaged.

At this juncture, if we ask value of `w`, the evaluator will force the thunk, namely, evaluate the argument `(id 10)`. This again increments `count` and sets it to be 2 and return 10 to the procedure calls it—the outer `id`. The latter one simply returns whatever it accepted, since the assignment expression has been evaluated the moment we defined `w`.

```
;;; L-Eval input:
w
```

Email address: informlarry@gmail.com

[1] This exercise demonstrates that the interaction between lazy evaluation and side effects can be very confusing. This is just what you might expect from the discussion in chapter 3.

```
;;; L-Eval value:
10

;;; L-Eval input:
count

;;; L-Eval value:
2
```