Exercise 1.41.

Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

Answer.

Instructed by the discription given above, we can easily implement the procedure `double` as

```
(define double
  (lambda (f)
    (lambda (x)
      (f (f x)))))
```

As is indicated by the substitution model, in order to evaluate the expression

```
(((double (double double)) inc) 5)
```

the interpreter will first evaluate subexpressions in any order, that is, `((double (double double)) inc)` and `5`. Since `5` is a number which is self-evaluating, so the interpreter now only has to evaluate the operator `((double (double double)) inc)` in this combination, then substitute `5` for the corresponding procedure parameter in the body of the procedure.

Note that we have been informed of the value of `inc`, which is is a procedure that adds 1 to its argument. Hence, the key to obtaining the value of the compound procedure `((double (double double)) inc)` lies in evaluating its operator, which is `(double (double double))`.

So far, one might rashly substitute

```
(lambda (f)
  (lambda (x)
    (f (f x))))
```

for `double` in the inner-most of `(double (double double))`. Unfortunately, this hasty practice will invlove a horrible amount of computation, thus cause that guy giddy a lot.

To controll its computational complexity, we'd better perform abstraction on `(double (double double))` linguistically. Notice that it is the subexpression `(double double)` which is awkward that keeps us annoying in evaluating `(double (double double))`. So we hope to express it with a more abstract but concise operation

```
(define quadruple (double double))
```

where the compound procedure `quadruple` which applies the original procedure four times can also be described as

```
(define quadruple
  (lambda (f)
    (lambda (x)
      (f (f (f (f x)))))))
```

Now we are able to reduce the expression `(double (double double))` to

```
(double quadruple)
```

Hence, the operator at the very beginning expression can be expressed as

```
((double quadruple) inc)
```

which evolves into

```
(quadruple (quadruple inc))
```

This indicates that *do-inc-four-times* will be performed four times, that is, *do-inc-sixteen-times*. So, the original combination turns out to be a procedure which takes a number (which is 5 here), then adds 16 to it. Therefore, what eventually return `(((double (double double)) inc) 5)` is `21`.