Exercise 2.6.

In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))

(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as *Church numerals*, after its inventor, Alonzo Church, the logician who invented the λ calculus.

Define `one` and `two` directly (not in terms of `zero` and `add-1`). (Hint: Use substitution to evaluate `(add-1 zero)`). Give a direct definition of the addition procedure `+` (not in terms of repeated application of `add-1`).

Answer.

As is indicated in the problem discription, to represent `one` here in the Church numeral, just evaluate `(add-1 zero)` by means of substitution:

```
(add-1 zero)
(lambda (f)
  (lambda (x)
    (f ((zero f) x))))
(lambda (f)
  (lambda (x)
    (f (((lambda (f)
           (lambda (x) x))
          f)
         x))))
(lambda (f)
  (lambda (x)
    (f ((lambda (x) x) x))))
(lambda (f)
  (lambda (x)
    (f x)))
```

Therefore, `one` can be expressed in terms of procedure as:

```
(define one
  (lambda (f)
    (lambda (x)
      (f x))))
```

Further more, the procedural definition of `two` can also be obtained by evaluating `(add-1 one)`:

```
(define two
  (lambda (f)
    (lambda (x)
      (f (f x)))))
```

Now, let's take up to bring the definition of the addition procedure `+` to the surface. Since the addition operation `+` is performed on integers, we'd better clarify a question at first: what is the general representation of an integer here in the Church numerals?

Well, by investigating the shapes of procedural definition of `zero`, `one` and `two`, we see that the successor of an integer is obtained by wrapping an additional `f` around the integer itself. Hence, an arbitrary integer n can be represented by a procedure in which we successively applying `f` to `x` for n times: $f\,(f\,(...(f\,(x))...))$. But, it is unacceptable to implement this idea in the program by embracing `f` to its former result over and over again.

On the other hand, while following the process generated in evaluating `zero` and `add-1`, we find that `zero` is defined to be a procedure which takes as its argument a procedure and returns as its value a procedure. The procedure generated in evaluating `(zero f)` takes in an object and returns that object invariantly. By comparision, the procedure `add-1` is established on a relatively higher level where it manipulates procedurally defined numerals like `zero`. Whenever `(add-1 n)` is evaluated, the underlying expression `((n f) x)` generates a procedure to represent the number n in our sence. This inspires us to represent the number n procedurally as:

```
(define n
  (lambda (f)
    (lambda (x)
      ((n f) x))))
```

Now, let's try to express the idea of addition by Church numerals. The former analysis on how `zero` and `(add-1 n)` work has given us some intuition in doing this. We saw that an arbitrary nonnegtive integer n is expressed by repeatedly applying `f` to `x` for n times. On the other hand, we can express n mathematically as $n = n + 0$. So in any case, the expression `((n f) x)` generates a procedure to represent the number n as well as adding n to 0.

Suppose that we want to add up two nonnegtive integers, say, m and n in terms of Church numerals. Using the evaluation rule above we've just discovered, the operation adding m to n should be expressed by a procedure which repeatedly applise `f` to `n` for m times. This reveals the definition of the addition procedure `+`:

```
(define (+ m n)
  (lambda (f)
    (lambda (x)
      ((m f) ((n f) x)))))
```