Exercise 3.10.

In the `make-withdraw` procedure, the local variable `balance` is created as a parameter of `make-withdraw`. We could also create the local state variable explicitly, using `let`, as follows:

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

Recall from section 1.3.2 that `let` is simply syntactic sugar for a procedure call:

```
(let ((<var> <exp>)) <body>)
```

is interpreted as an alternate syntax for

```
((lambda (<var>) <body>) <exp>)
```

Use the environment model to analyze this alternate version of `make-withdraw`, drawing figures like the ones above to illustrate the interactions

```
(define W1 (make-withdraw 100))

(W1 50)

(define W2 (make-withdraw 100))
```

Show that the two versions of `make-withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?

Answer.

The procudure definition of `make-withdraw` above would have been equivalent to have used

```
(define (make-withdraw initial-amount)
  ((lambda (balance)
     (lambda (amount)
       (if (>= balance amount)
           (begin (set! balance (- balance amount))
                  balance)
           "Insufficient funds")))
   initial-amount))
```

Following the same way, let us describe the evaluation of

```
(define W1 (make-withdraw 100))
```

followed by

```
(W1 50)
50
```

as well as that of

```
(define W2 (make-withdraw 100))
```

Figure 1 shows the result of defining the `make-withdraw` procedure in the global environment. This produces a procedure object that contains some code together with a pointer to the global environment. It remain almost the same as the example above, except that the body of the procedure is an application rather than the `lambda` expression.

To evaluate

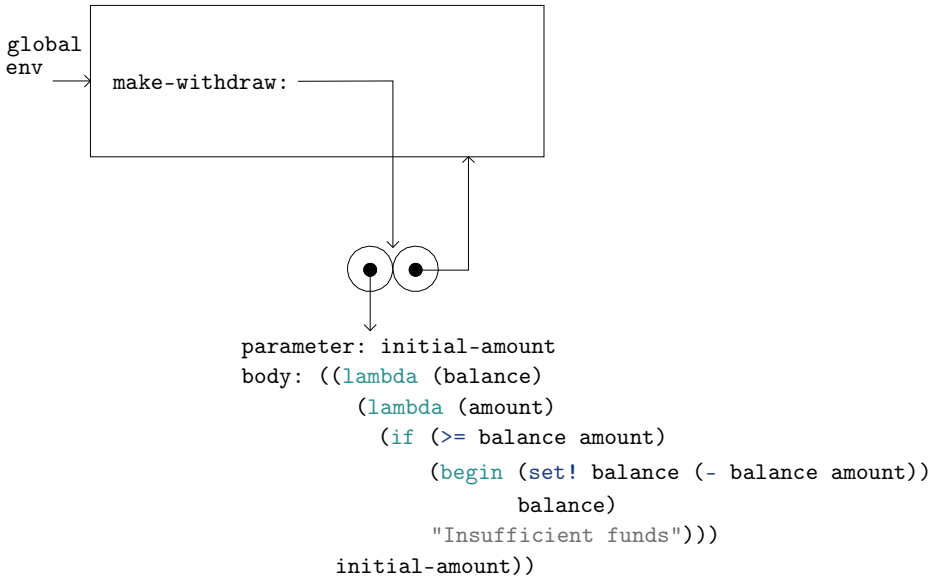

Figure 1. Result of defining the `make-withdraw` procedure in the global environment.

```
(define W1 (make-withdraw 100))
```

We follow the same way before by setting up an environment E1 in which the formal parameter `initial-amount` is bound to the argument 100. Within this environment, we evaluate the body of `make-withdraw`, namely the application. This establish a new environment E2 in which `balance`, the formal parameter of the outer `lambda` expression, is bound to `initial-amount`, which is 100. Figure 2 shows the intermediate environment structure.

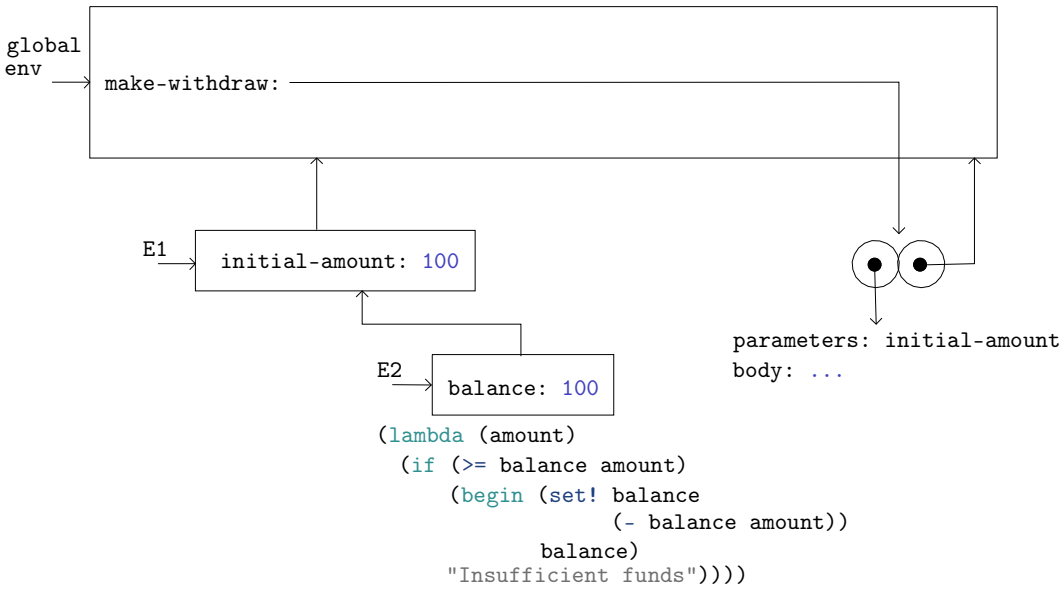

Figure 2. Intermediate environment structure in evaluating `(define W1 (make-withdraw 100))`.

Within the environment E2, we evaluate the body of the resulting procedure, which is also a `lambda` expression but takes as its argument the variable `amount`. This constructs a new procedure object, whose code is as specified by the `lambda` and whose environment is E2, the environment in which the `lambda` was evaluate to produce the procedure. That is the resulting procedure object returned by the call to `make-withdraw`. This is bound to `W1` in the global environment, for the `define` itself is being evaluated in the global environment. Figure 3 shows the resulting environment structure.

Now we can analyze when `W1` is applied to an argument:

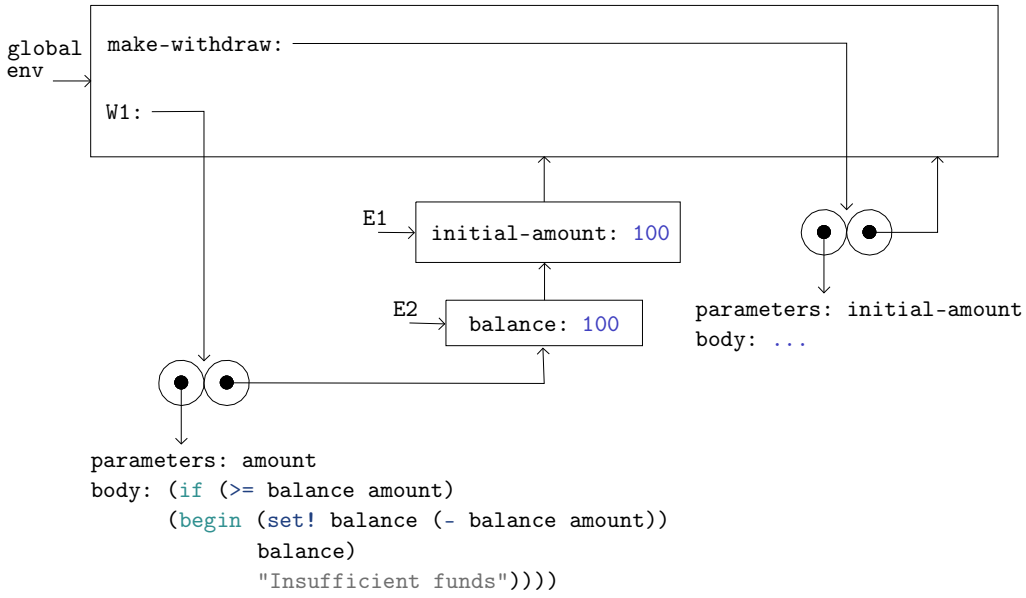

Figure 3. Result of evaluating `(define W1 (make-withdraw 100))`.

```
(W1 50)
50
```

We begin by constructing a frame in which `amount`, the formal parameter of `W1`, is bound to the argument 50. Similarly, this frame has as its enclosing environment not the global environment, but rather the environment E2, because this is the environment that is specified by the `W1` procedure object. Within this new environment, we evaluate the body of the procedure:

```
(if (>= balance amount)
    (begin (set! balance (- balance amount))
           balance)
    "Insufficient funds")
```

The resulting environment structure is shown in figure 4. The expression being evaluated references both `amount` and `balance`. `Amount` will be found in the first frame in the environment, while `balance` will be found by following the enclosing environment pointer to E2.

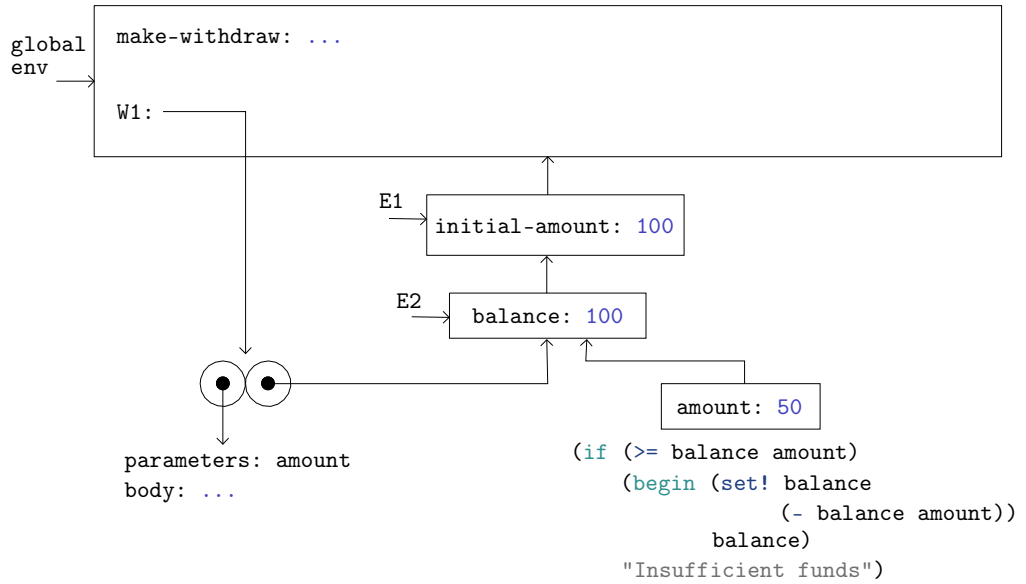

Figure 4. Environment created by applying the procedure object `W1`.

When the `set!` is executed, the binding of `balance` in E2 is changed. At the completion of the call to `W1`, `balance` is 50, and the frame that contains `balance` is still pointed to by the procedure object `W1`. The frame that binds `amount` (in which we executed the code that changed `balance`) is no longer relevant, since the procedure call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment. The next time `W1` is called, this will build a new frame that binds `amount` and whose enclosing environment is E2. We see that E2 serves as the "place" that holds the local state variable for the procedure object `W1`. Figure 5 shows the situation after the call to `W1`.

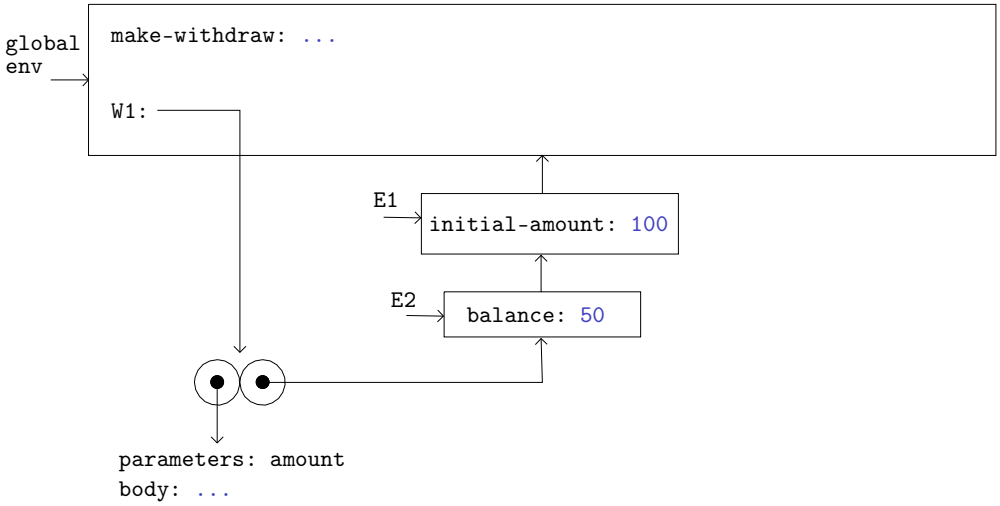

Figure 5. Environment after the call to `W1`.

When we create a second call "withdraw" object by making another call to `make-withdraw`:

```
(define W2 (make-withdraw 100))
```

This produces the environment structure of figure 6, which shows that `W2` is a procedure object, that is, a pair with some code and an environment. The environment E2 for `W2` was created by the call to `make-withdraw`. It contains a frame with its own local binding for `balance`.

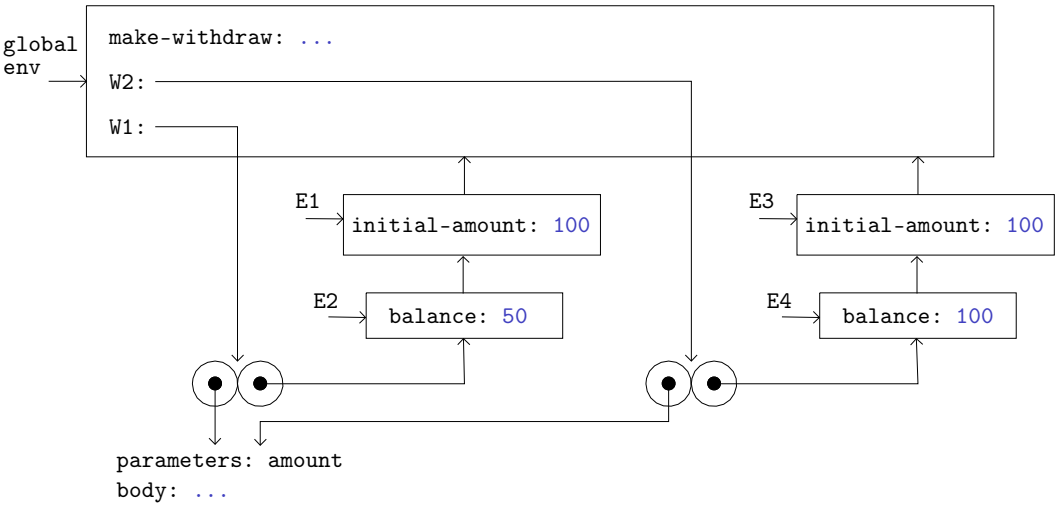

Figure 6. Using `(define W2 (make-withdraw 100))` to create a second object.