Exercise 3.10

Problem statement

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 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?

Solution

Knowing what let is actually syntactic sugar for, we can redefine `make-withdraw` as

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

Remember the original `make-withdraw` was declared as

```
(define (make-withdraw balance)
  (lambda (amount)
```

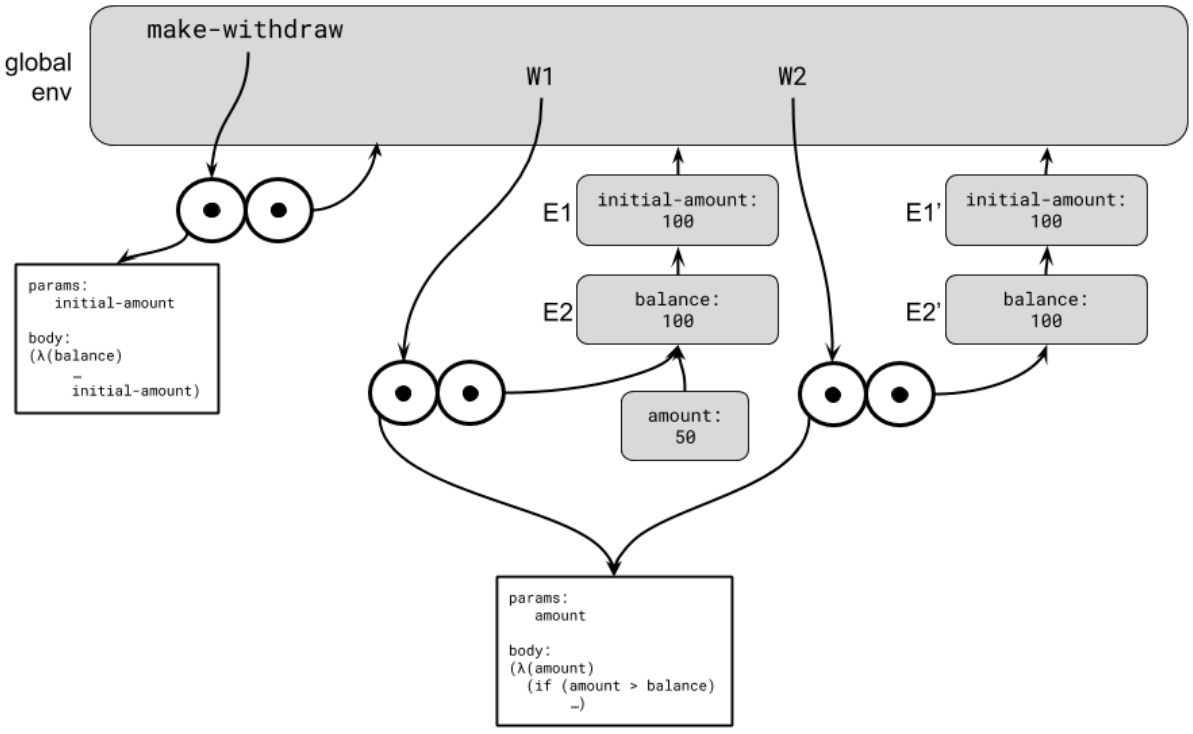

Figure 1: Frame structure for the calls to `make-withdraw`.

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

The two expressions are functionally equivalent. Essentially the new double λ version defines an extra anonymous function that it then applies to the balance given. This results in one extra environment being created that holds the `initial-amount` value. The anonymous λ is applied on `initial-amount` from environment E1. This generates a new environment, E2 that points to E1, and contains the value for `balance`. `W1` is bound to a procedure object that points to the environment E2.

When we make the call `(W1 50)` a frame is created that points to E2. The value of balance is modified in E2. Every time we call `W1` a new frame that points to E2 will be created and modify the value of balance if the value we call `W1` with is smaller than the remaining balance.

When we create `(define W2 (make-withdraw 100))` a new stream of environments will be created. Lets call the new frame that holds an `initial-amount` value E1', and the one that holds the `balance` value E2'. Figure 1 shows the frame structure created by the calls to `make-withdraw`.