DDG University

Informal Goals

1. Learn new things related to technology.
2. Learn from each other.
3. Foster inter-team building.
4. To become better engineers.

Search for DDG University in Asana.

Structure and Interpretation of Computer Programs *(SICP)*

by Harold Abelson and Gerald Jay Sussman

3.2 The Environment Model of Evaluation

1. ***Exercises***

Exercise 3.9

In 1.2.1 we used the substitution model to analyze two procedures for computing factorials, a recursive version

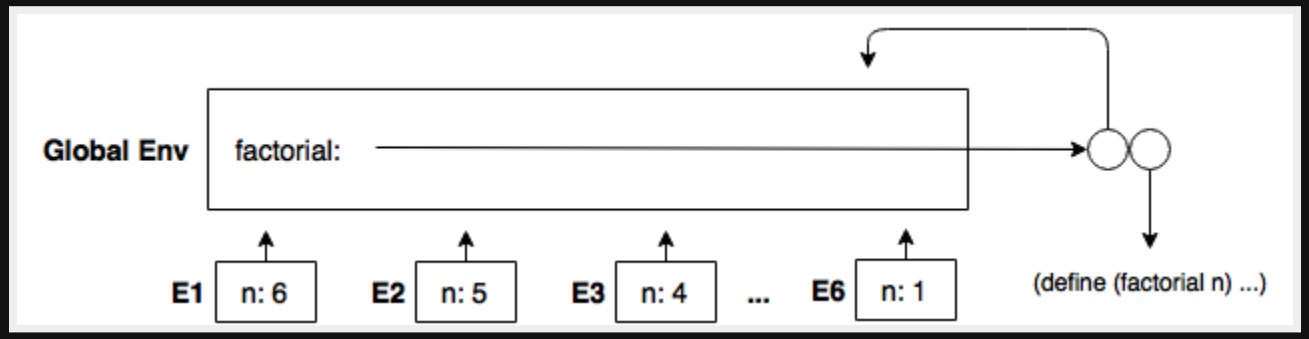
```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

and an iterative version

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product
          counter
          max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
           (+ counter 1)
           max-count)))
```

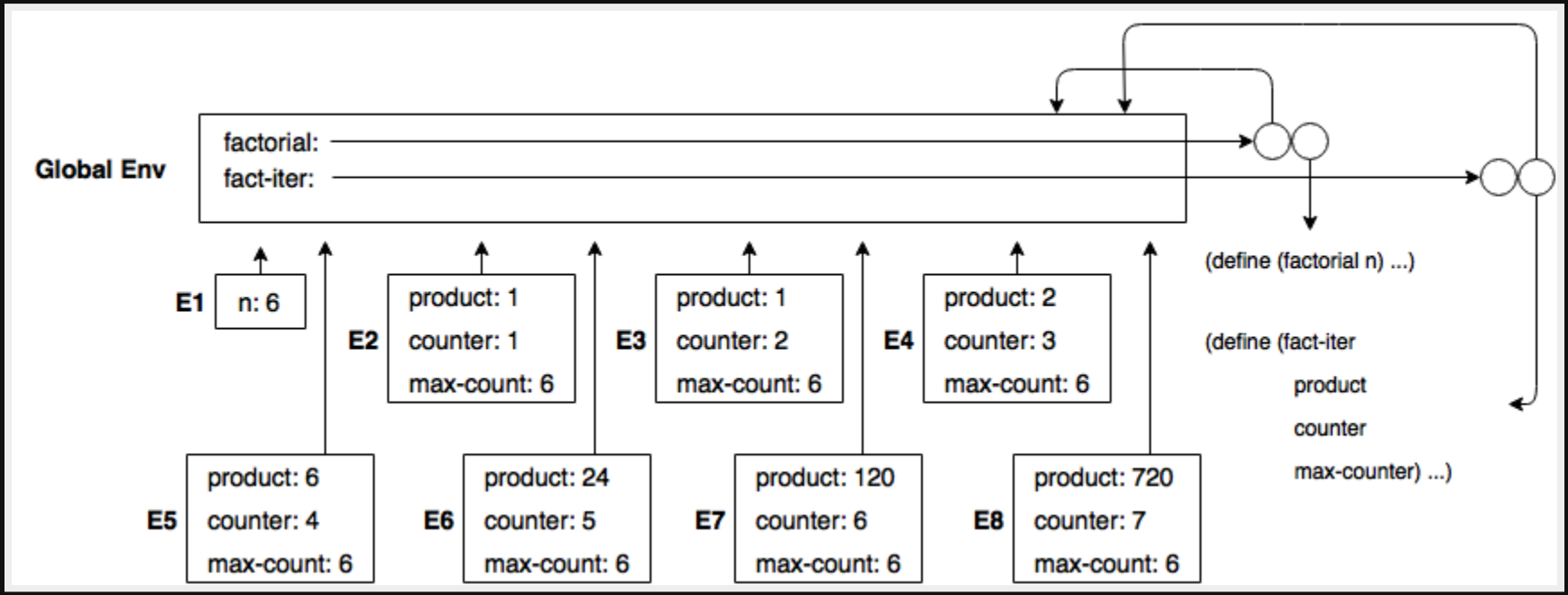
Show the environment structures created by evaluating `(factorial 6)` using each version of the factorial procedure.

Exercise 3.9 *(cont)*




```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Exercise 3.9 *(cont)*




```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product
          counter
          max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
          (+ counter 1)
          max-count)))
```

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 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⟩)
```

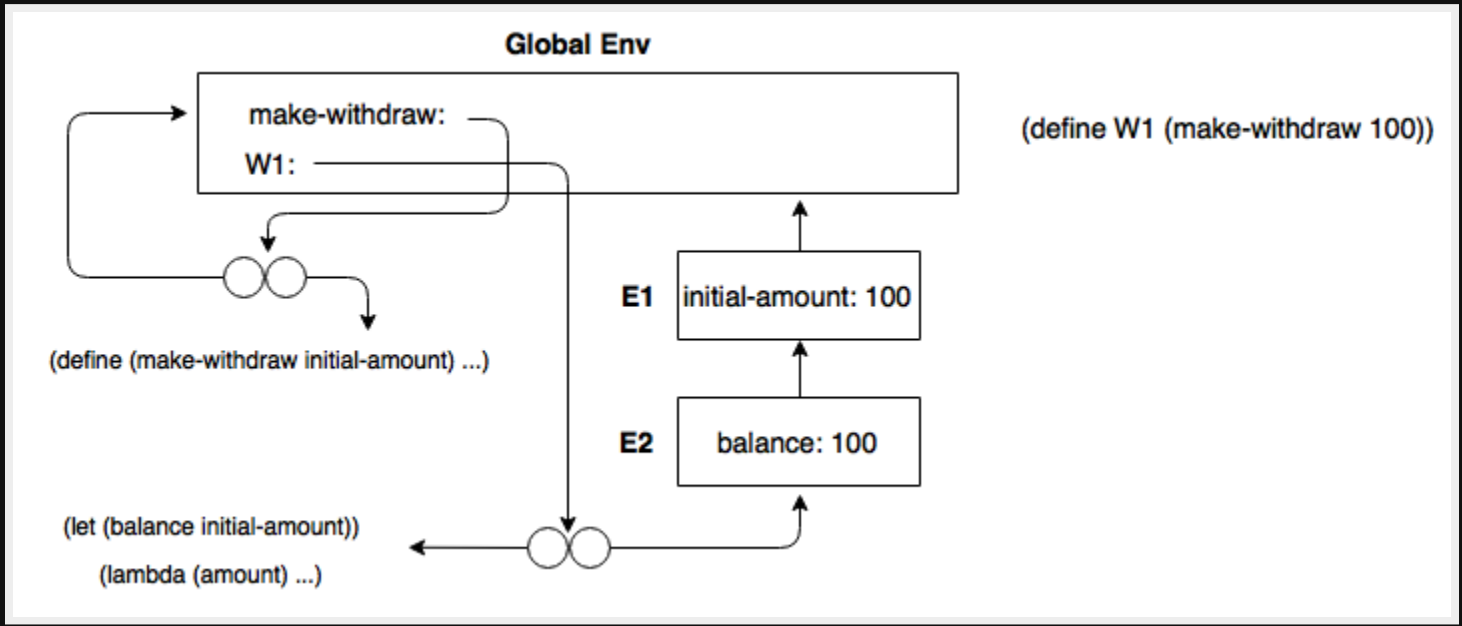
Exercise 3.10 *(cont)*

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))
```

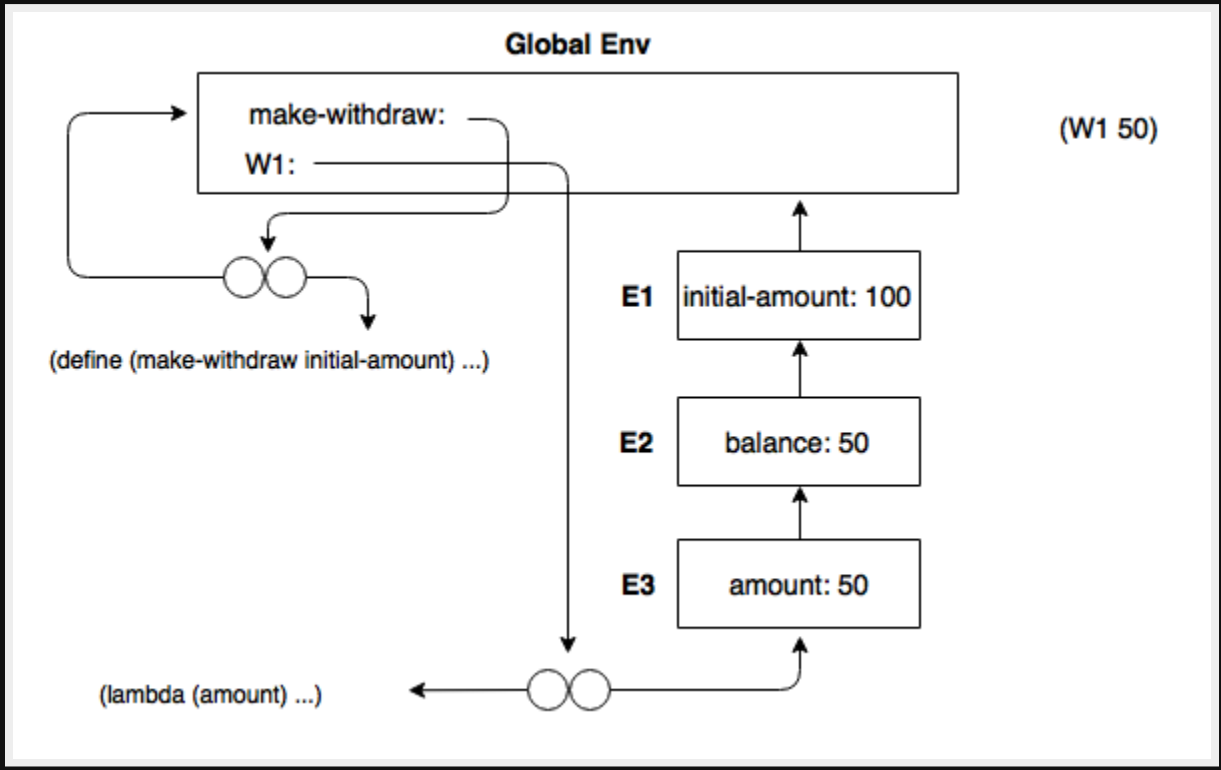
Show that the two versions of make-withdraw create objects with the same behavior. How do the environment structures differ for the two versions?

Exercise 3.10 *(cont)*



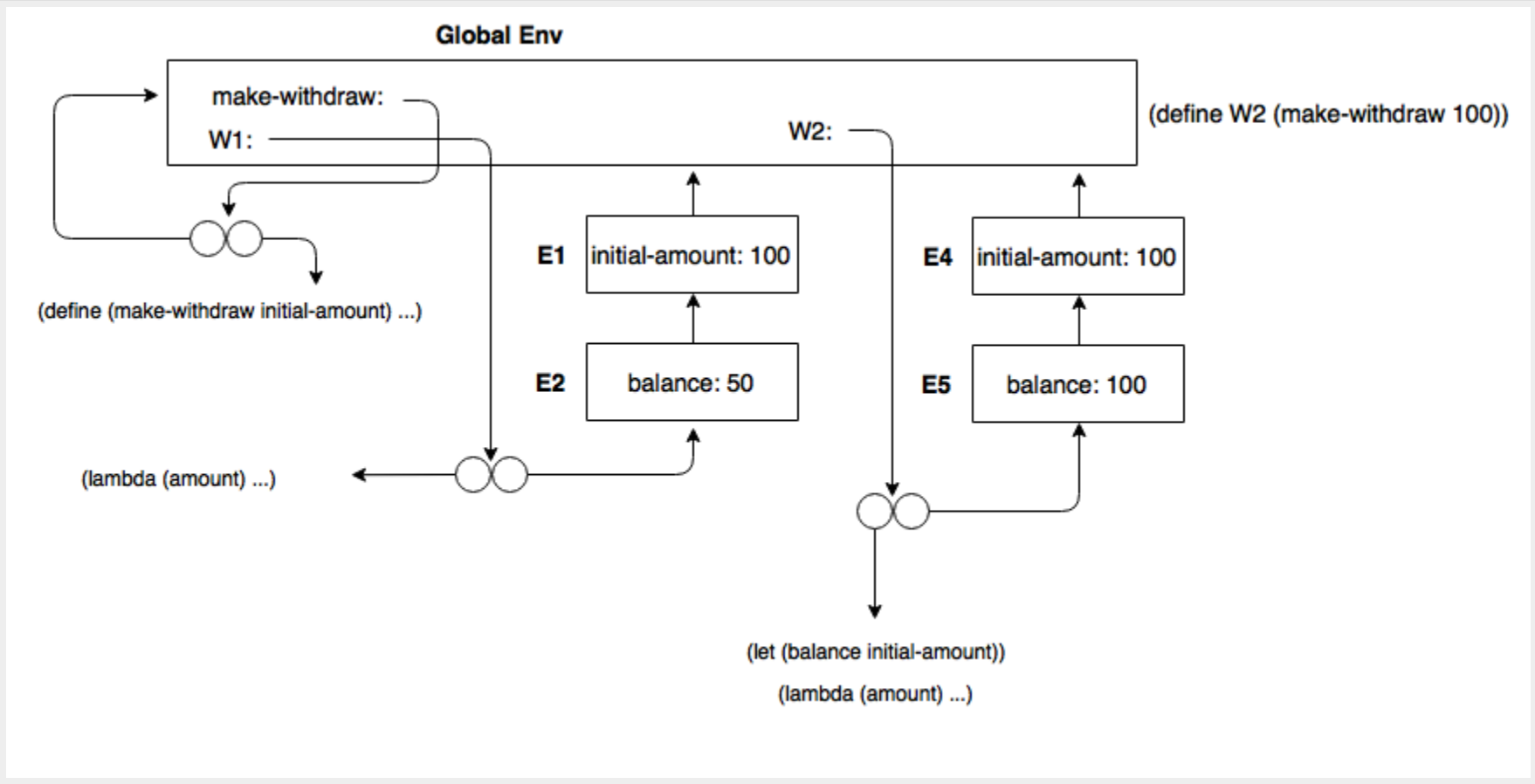

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance
           (- balance amount))
           balance)
          "Insufficient funds"))))
```

Exercise 3.10 *(cont)*




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

Exercise 3.10 *(cont)*




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

Exercise 3.11

In 3.2.3 we saw how the environment model described the behavior of procedures with local state. Now we have seen how internal definitions work. A typical message-passing procedure contains both of these aspects. Consider the bank account procedure of 3.1.1:

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

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)
```

Exercise 3.11 *(cont)*

Show the environment structure generated by the sequence of interactions:

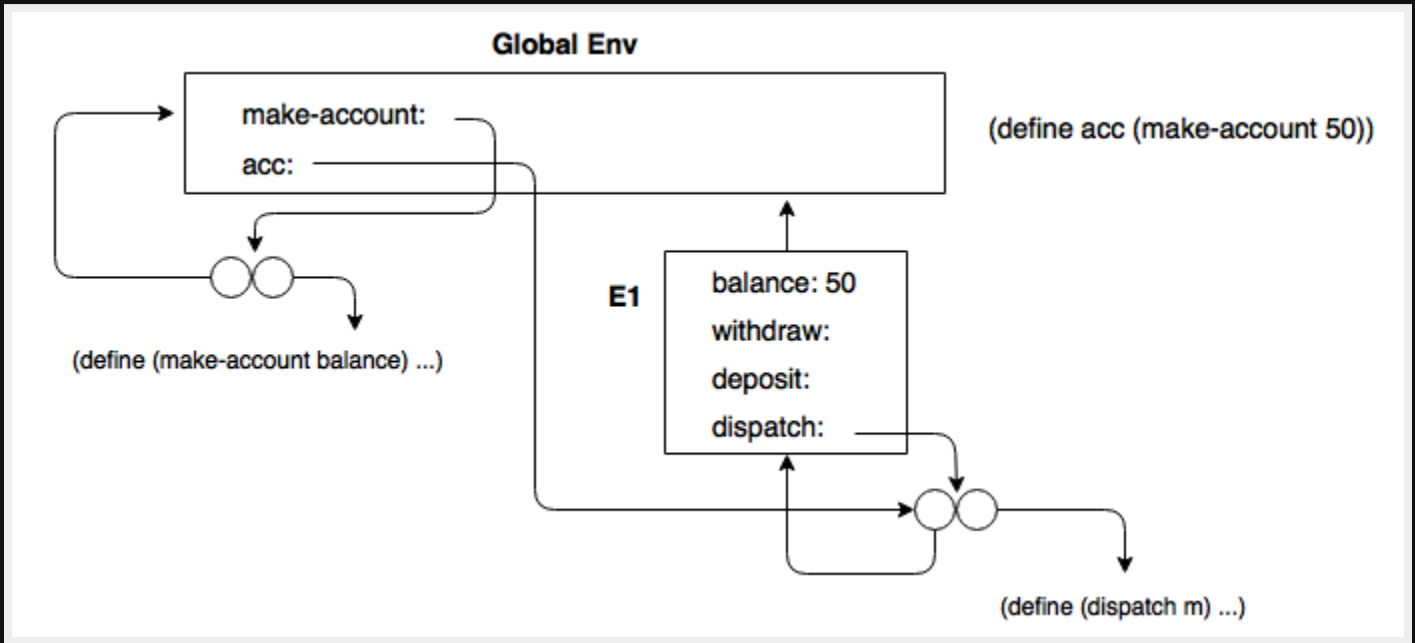
```
(define acc (make-account 50))

((acc 'deposit) 40)
90

((acc 'withdraw) 60)
30
```

Where is the local state for acc kept? Suppose we define another account

```
(define acc2 (make-account 100))
```

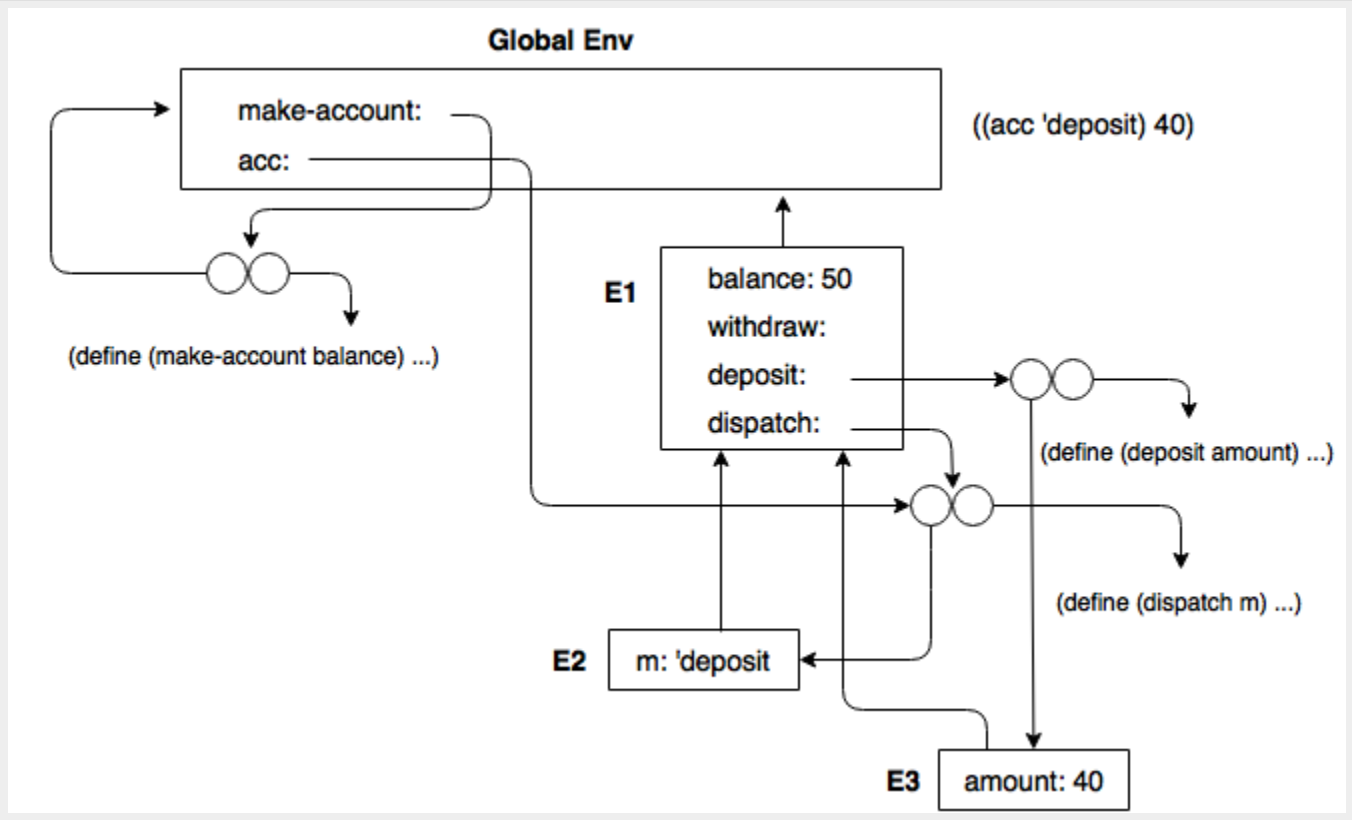
How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between acc and acc2?

Exercise 3.11 *(cont)*




```
(define (make-account balance)
  (define (withdraw amount) ...)

  (define (deposit amount) ...)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)
```

Exercise 3.11 *(cont)*




```
(define (make-account balance)
  (define (withdraw amount) ...)

  (define (deposit amount) ...)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)
```

That's all for section 3.2.

Thanks!