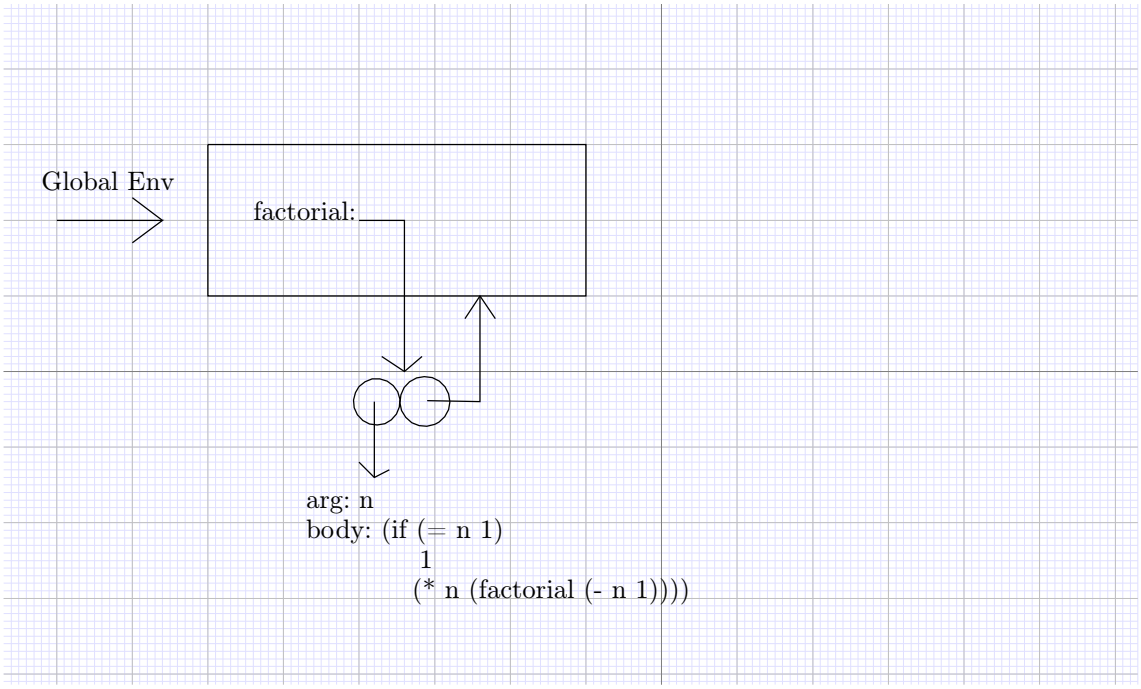
Chapter 3 - Drawing Exercise

by Bill Xue

Exercise 3.9

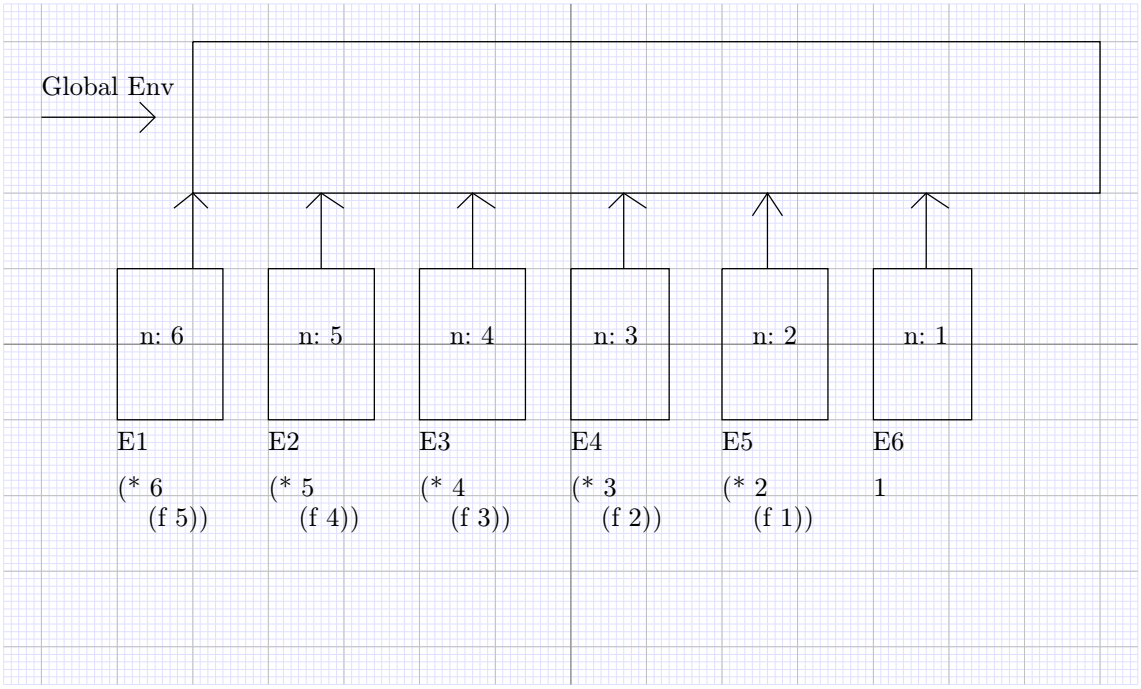
factorial recursive version:

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



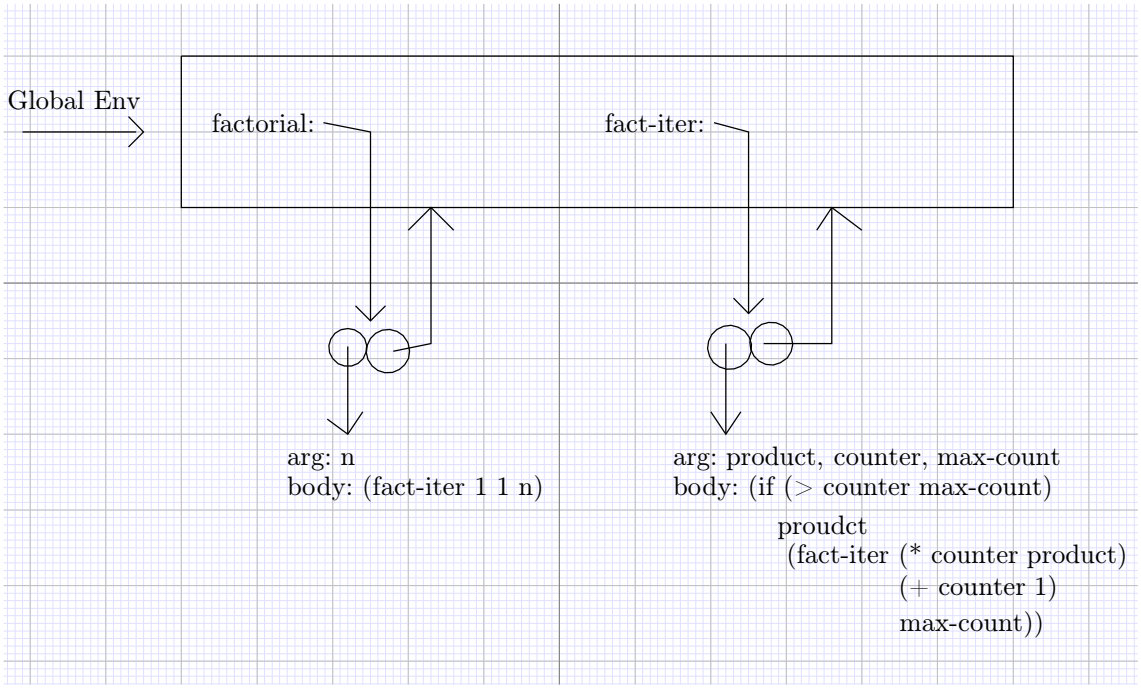
```
Scheme] (factorial 6)
```

Using f representing *factorial*



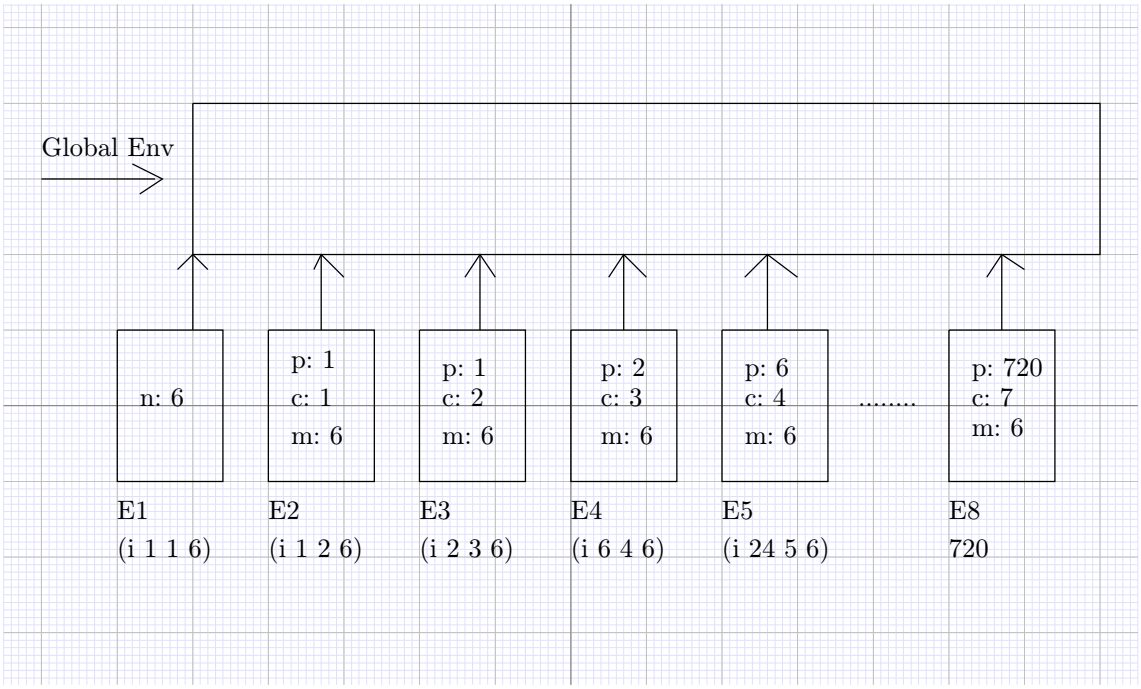
factorial iteration version

```
Scheme] (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)))
```




```
Scheme] (factorial 6)
```

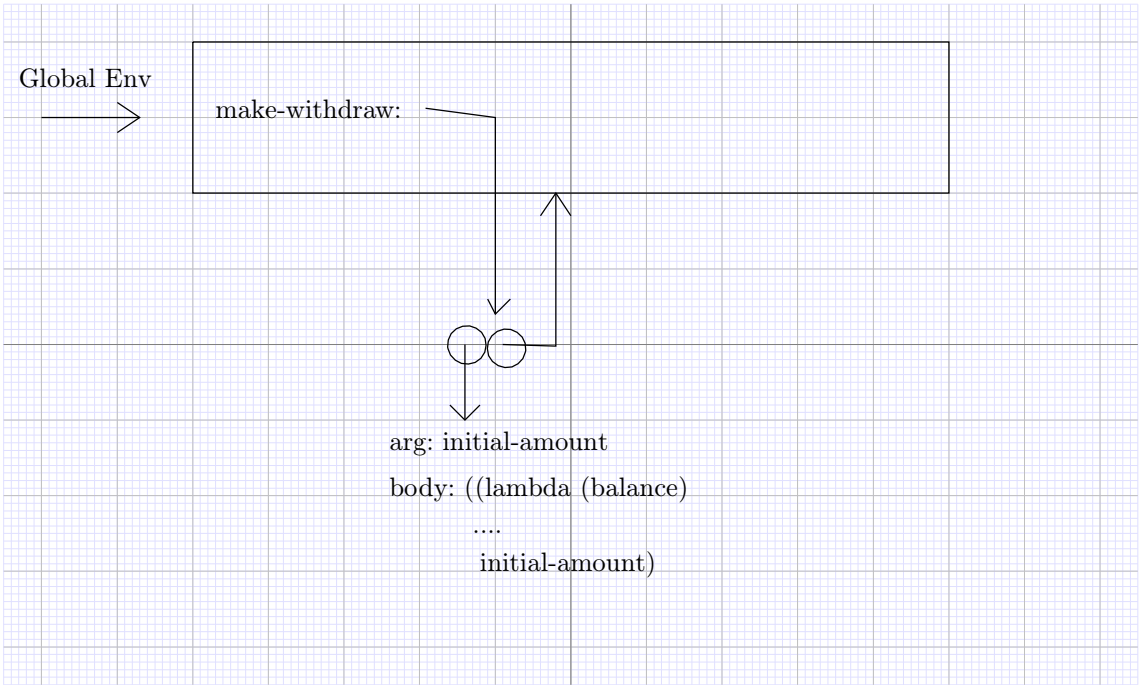
Using f representing *factorial*, i for *fact-iter* p for *product*, c for *counter*, m for *max-count*



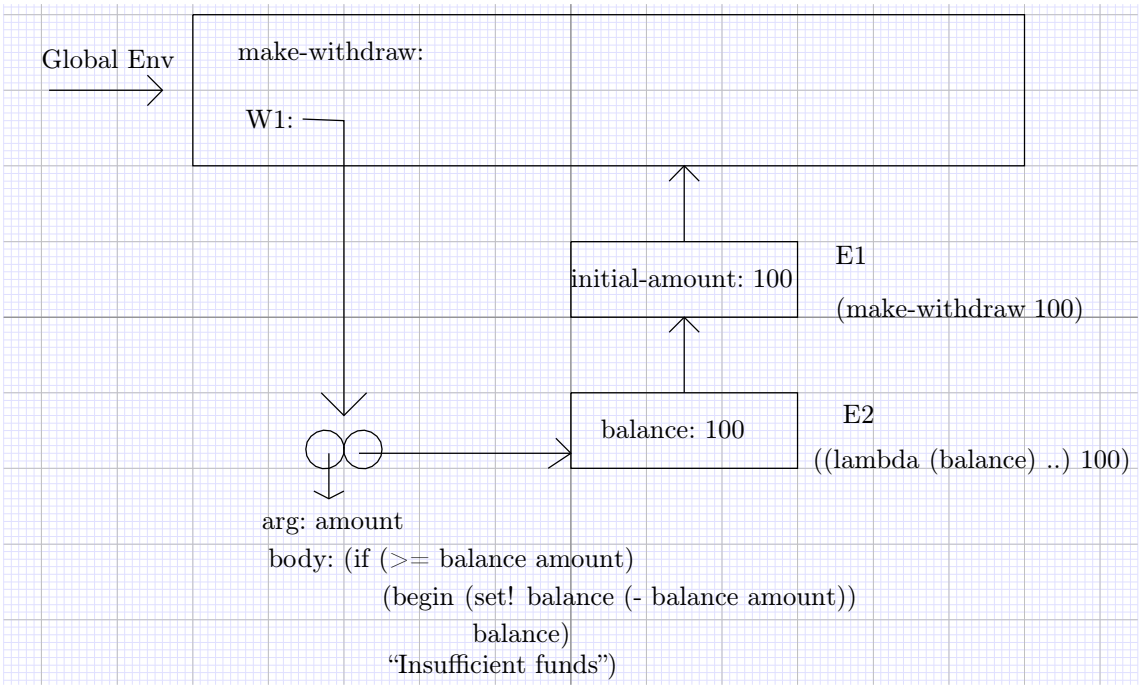
Exercise 3.10

After translating *let* into *lambda*

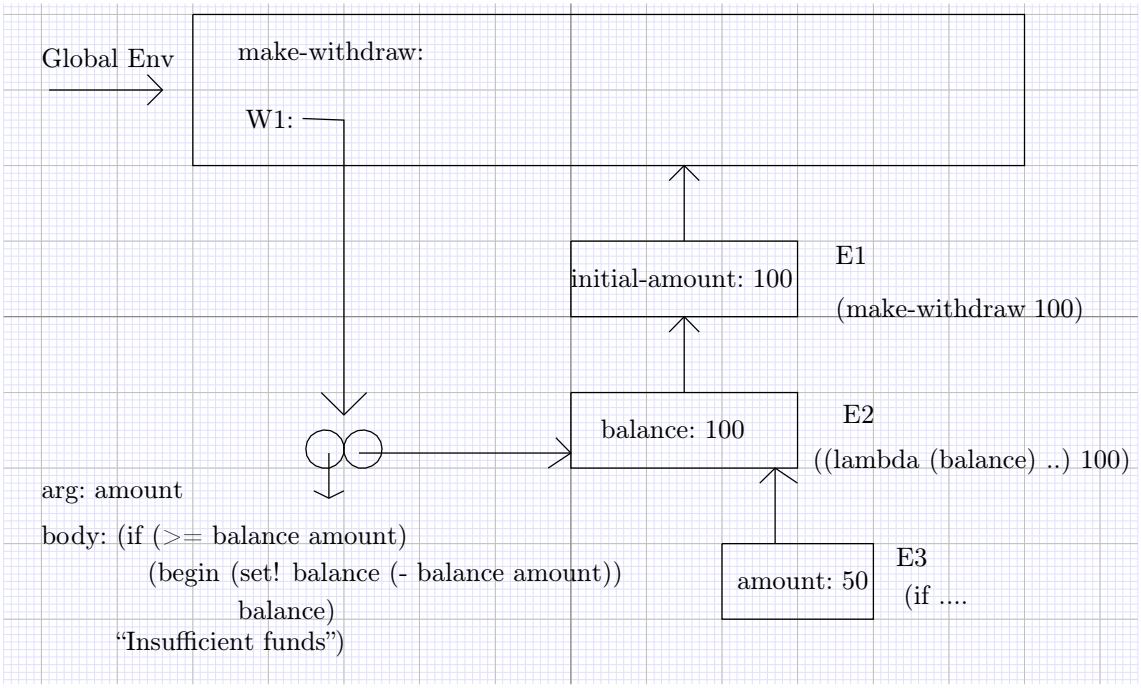
```
Scheme] (define make-withdraw
          (lambda (initial-amount)
            ((lambda (balance)
               (lambda (amount)
                 (if (>= balace amount)
                     (begin (set! balance (- balance amount))
                            balance)
                     "Insufficient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




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



```
Scheme] (W1 50)
```




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

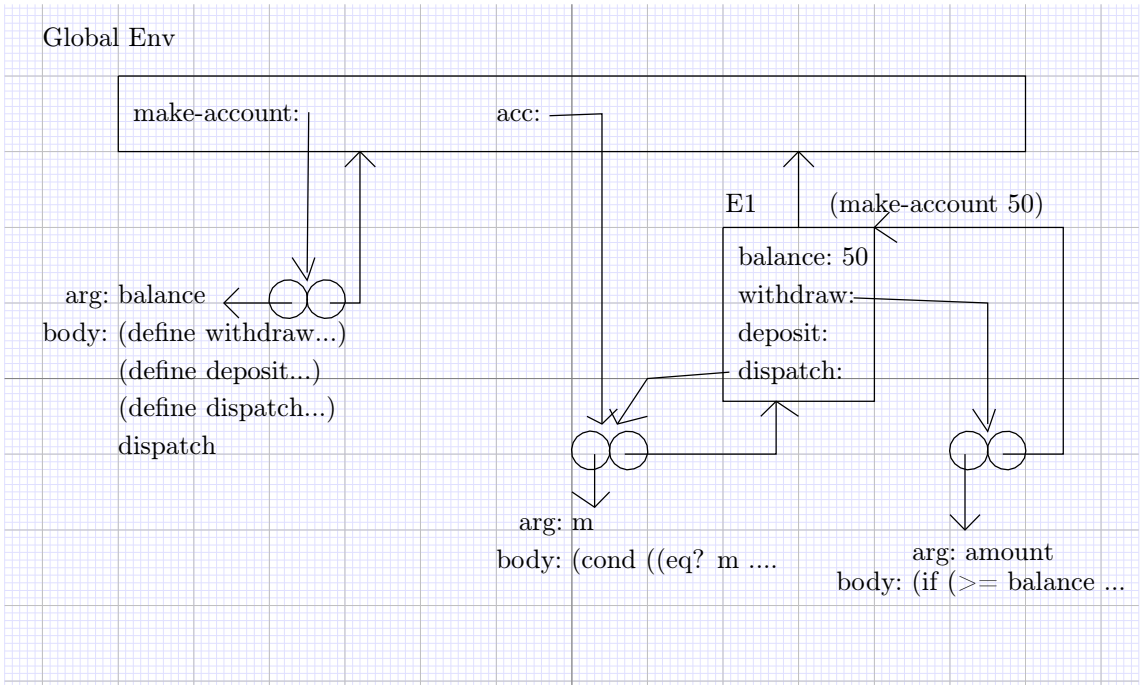
It makes almost a copy of *W1*, which has its own *E1*, *E2*

Exercise 3.11

```
Scheme] (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)
```

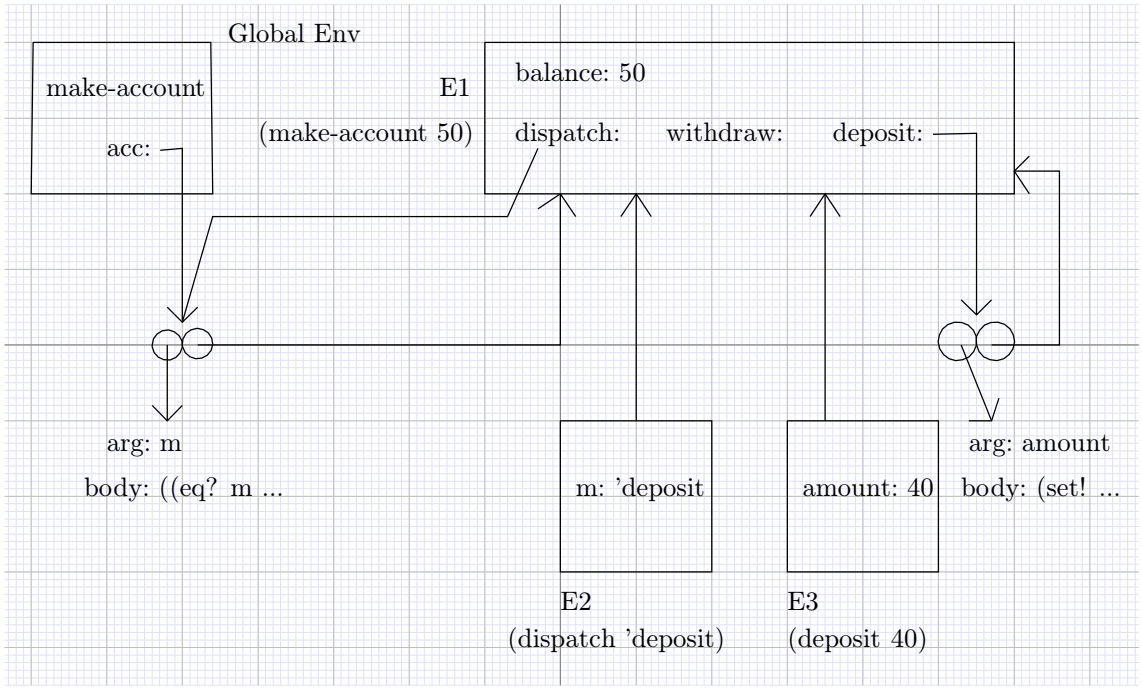
```
Scheme] (define acc (make-account 50))
```

Currently, the environment model is (procedure binded to *deposit* is omitted in *E1*):




```
Scheme] ((acc 'deposit) 40)
```

```
90
```

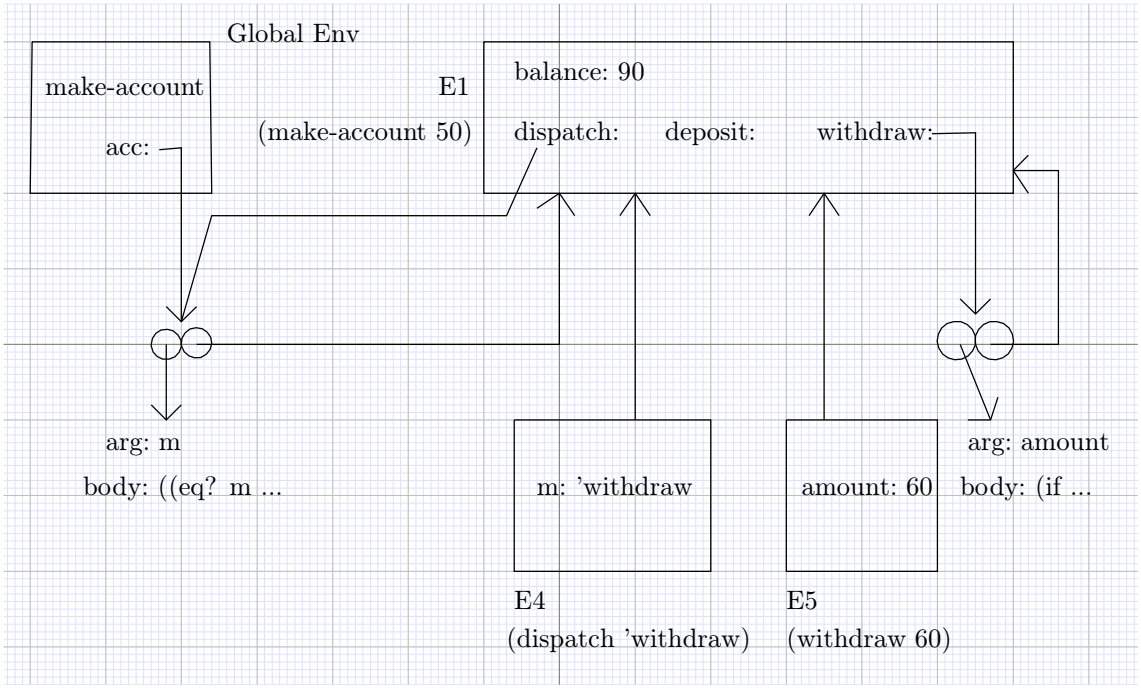



```
Scheme] ((acc 'withdraw) 60)
```

```
30
```

Currently, the *balance* in *E1* has been updated to 90.

When given the parameter m to *acc*, it generates a new environment *E4*, not to update *E2*.



As we can see, the local state of *acc* is stored in *balance* of *E1*

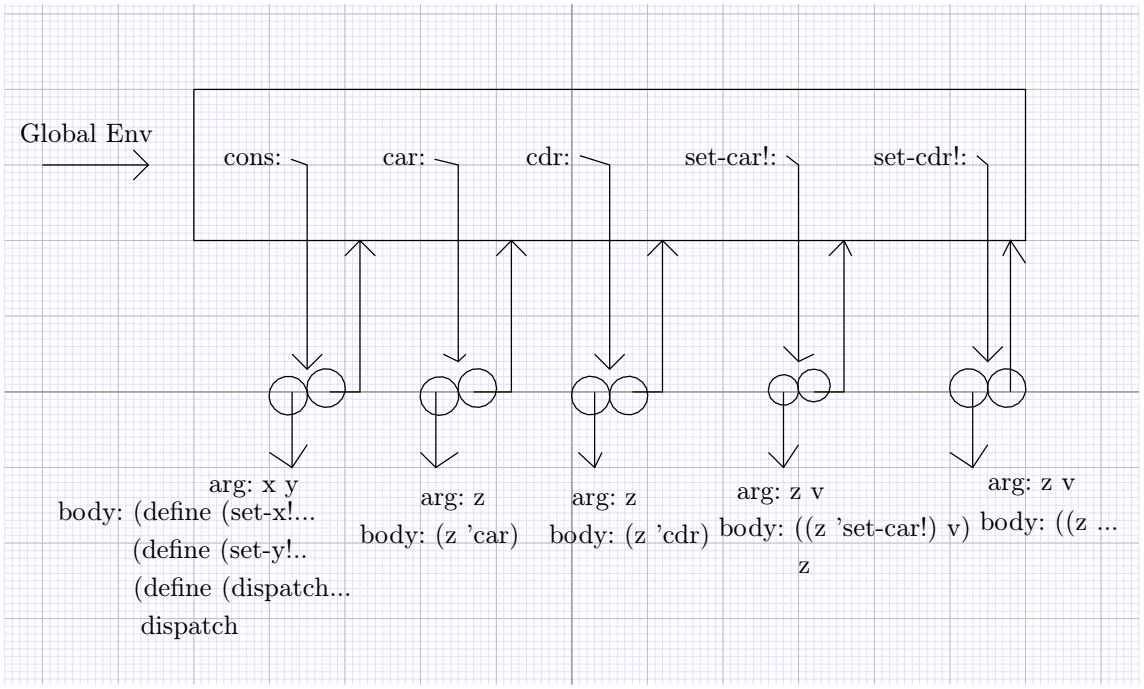
```
Scheme] (define acc2 (make-account 100))
```

The *acc2* doesn't share any procedures and variables with *acc*

Exercise 3.20

```
Scheme] (define (cons x y)
          (define (set-x! v) (set! x v))
          (define (set-y! v) (set! y v))
          (define (dispatch m)
            (cond ((eq? m 'car) x)
                  ((eq? m 'cdr) y)
                  ((eq? m 'set-car!) set-x!)
                  ((eq? m 'set-cdr!) set-y!)
                  (else
                    (error "Undefined operation -- CONS" m))))
          dispatch)
Scheme] (define (car z) (z 'car))
Scheme] (define (cdr z) (z 'cdr))
Scheme] (define (set-car! z new-value)
          ((z 'set-car!) new-value)
          z)
Scheme] (define (set-cdr! z new-value)
          ((z 'set-cdr!) new-value)
          z)
```

Currently, the environment model is:

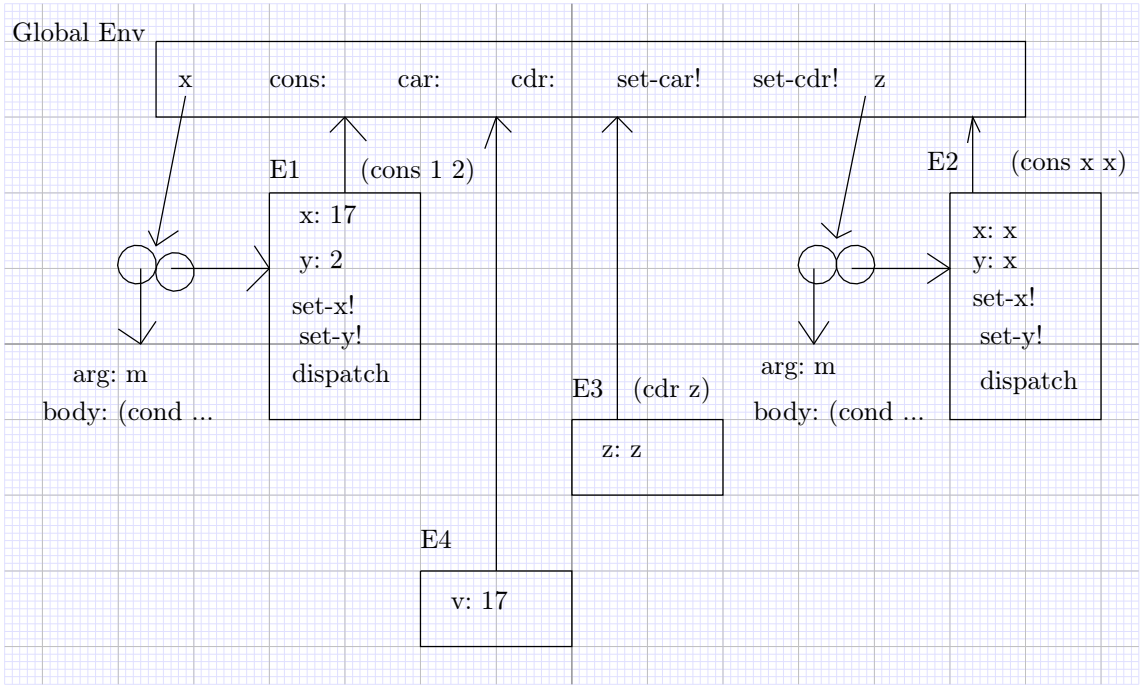



```
Scheme] (define x (cons 1 2))
Scheme] (define z (cons x x))
Scheme] (set-car! (cdr z) 17)
```

```
#<procedure x (m)>
```

```
Scheme] (car x)
```

```
17
```



How to handle the relationship between *E3* and *E4*? How to represent temp var *(cdr z)*?