# Contents



# 1 Modularity, Objects, and State

## 1.1 Assignment and Local State

### Exercise 3.1

> Write a procedure `make-accumulator` that generates accumulators, each maintaining an independent sum. The input to `make-accumulator` should specify the initial value of the sum.

We will use `set!` to mutate the current accumulator `sum`. After modifying the state we simply return the current value.

```
(define (make-accumulator sum)
  (lambda (n)
    (set! sum (+ sum n))
    sum))
```

Now we can test our procedure by creating an accumulator that adds 5 to the current sum.

```
(define A (make-accumulator 5))
(= (A 10) 15)
(= (A 10) 25)
```

**Exercise 3.2**

> Write a procedure `make-monitored` that takes as input a procedure, `f`, that itself takes one input. The result returned by `make-monitored` is a third procedure, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the special symbol `how-many-calls?`, then `mf` returns the value of the counter. If the input is the special symbol `reset-count`, then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter.

Let `calls` be a state variable for `mf` to mutate after each time `f` is called. The user can pass in the special symbol `how-many-calls?` to `mf` to see how many times `f` has been called.

```
(define (make-monitored f)
  (let ((calls 0))
    (define (mf arg)
      (cond ((eq? arg 'how-many-calls?) calls)
            ((eq? arg 'reset-count) (set! calls 0) 0)
            (else (set! calls (+ calls 1))
                  (f arg))))
    mf))
```

We copy the test from the book to ensure our `make-monitored` works as expected. In addition we show that resetting the count also works as intended.

```
(define s (make-monitored sqrt))
(s 100)
(= (s 'how-many-calls?) 1)
(s 'reset-count 0)
(= (s 'how-many-calls?) 0)
```

### Exercise 3.3

> Modify the `make-account` procedure so that it creates password-protected accounts.

Introduce a new state variable `password` where we store the user's password. It will be bound in the local environment from the arguments in `make-account`. Install a new procedure `wrong-password?` in the dispatch that checks against it. Handle incorrect passwords by calling `handle-incorrect-password`.

```
(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (begin (set! balance (+ balance amount))
           balance))
  (define (handle-incorrect-password x)
    "Incorrect password")
  (define (wrong-password? pwd)
    (not (eq? pwd password)))
  (define (dispatch pwd m)
    (cond ((wrong-password? pwd) handle-incorrect-password)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)
```

Again the test from the book to verify our procedure.

```
(define acc (make-account 100 'secret-password))
(= ((acc 'secret-password 'withdraw) 40) 60)
(string=? ((acc 'some-other-password 'deposit) 50) "Incorrect password")
```

### Exercise 3.4

> Modify the make-account procedure of `Exercise 3.3` by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the procedure `call-the-cops`.

The plan is to introduce a local state variable `incorrect-attempts` that holds the number of password attempts. We then modify `handle-incorrect-password` so that it increments the local state. If the number of attempts is more than 7 we simply call the cops, otherwise just output `Incorrect password` as usual.

```scheme
(define (make-account balance password)
  (define incorrect-attempts 0)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
    (define (deposit amount)
      (begin (set! balance (+ balance amount))
             balance))
  (define (signal-incorrect-password)
    "Incorrect password")
  (define (call-the-cops)
    "Calling the cops")
  (define (handle-incorrect-password x)
    (begin (set! incorrect-attempts (+ incorrect-attempts 1))
           (if (>= incorrect-attempts 7)
               (call-the-cops)
               (signal-incorrect-password))))
  (define (handle-request request)
    (set! incorrect-attempts 0)
    (cond ((eq? request 'withdraw) withdraw)
          ((eq? request 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" request))))
  (define (wrong-password? pwd)
    (not (eq? pwd password)))
  (define (dispatch pwd m)
    (if (wrong-password? pwd)
        handle-incorrect-password
        (handle-request m)))
  dispatch)
```

As a test we create an account and provide the wrong password 7 times to see that the cops are called.

```scheme
(define acc (make-account 100 'secret-password))
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
```

```
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
(string=? ((acc 'some-other-password 'deposit) 40) "Calling the cops")

(= ((acc 'secret-password 'deposit) 40) 140)
(string=? ((acc 'some-other-password 'deposit) 40) "Incorrect password")
```

Note that after writing the correct password the number of attempts is reset.

**Exercise 3.5**

> Implement Monte Carlo integration as a procedure `estimate-integral` that takes as arguments a predicate $P$, upper and lower bounds $x_1, x_2, y_1$ and $y_2$ for the rectangle, and the number of trials to perform in order to produce the estimate. Your procedure should use the same monte-carlo procedure that was used above to estimate $\pi$ . Use your `estimate-integral` to produce an estimate of $\pi$ by measuring the area of a unit circle.

Let's first copy procedures `monte-carlo` and `random-in-range` from the book.

```
(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1)
                 (+ trials-passed 1)))
          (else (iter (- trials-remaining 1)
                      trials-passed))))
  (iter trials 0))

(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

Now we can implement `estimate-integral`. We simply generate random points in the rectangle and check if they are in the unit circle. The ratio of points in the circle to the total number of points is an estimate of the area of the circle. Since the area of the circle is $\pi$ we can multiply the ratio by the area of the rectangle to get an estimate of $\pi$.

```
(define (estimate-integral P x1 x2 y1 y2 trials)
  (define (experiment)
    (let ((x (random-in-range x1 x2))
      (y (random-in-range y1 y2)))
      (P x y)))
  (* (monte-carlo trials experiment) (* (- x2 x1) (- y2 y1))))
```

To get our result we define a predicate `in-unit-circle?` that checks if a point is in the unit circle. We then call `estimate-integral` with the predicate and the bounds of the unit circle.

```
(define (in-unit-circle? x y) (<= (+ (square x) (square y)) 1))
(define pi-approx (estimate-integral in-unit-circle? -1.0 1.0 -1.0 1.0 1000))
```

### Exercise 3.6

> Design a new rand procedure that is called with an argument that is either the symbol **generate** or the symbol **reset** and behaves as follows: **(rand 'generate)** produces a new random number; **((rand 'reset) ⟨new-value⟩)** resets the internal state variable to the designated **⟨new-value⟩**.

Following the footnote instructions we create a simple **rand-update** with values for $a, b$ and $m$ chosen from Wikipedia's Linear congruential generator article.

```
(define (rand-update x)
  (let ((a 4)
        (b 1)
        (m 9))
    (remainder (+ (* a x) b) m)))
```

Using this we can implement **rand**. If the argument is **generate** we simply update the current state using **rand-update** and then return next value. If the argument is **reset** we set the state to the new value.

```
(define rand
  (let ((x 0))
    (define (generate-random)
      (begin (set! x (rand-update x)) x))
    (define (reset-seed new-value)
      (begin (set! x new-value) '()))
    (define (dispatch m)
      (cond ((eq? m 'reset) reset-seed)
            ((eq? m 'generate) (generate-random))
            (else (error "Need a symbol 'reset or 'generate"))))
    dispatch))
```

Let's test our procedure by generating a random number and then resetting the seed to see if we get the same number again.

```
(= (rand 'generate) 1)
(= (rand 'generate) 5)
((rand 'reset) 0)
(= (rand 'generate) 1)
(= (rand 'generate) 5)
```

**Exercise 3.7**

> Suppose that our banking system requires the ability to make joint accounts. Define a procedure `make-joint` that accomplishes this. `make-joint` should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the `make-joint` operation to proceed. The third argument is a new password. `make-joint` is to create an additional access to the original account using the new password.

Let's grab our solution from `Exercise 3.3`.

```scheme
(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (begin (set! balance (+ balance amount))
           balance))
  (define (handle-incorrect-password x)
    "Incorrect password")
  (define (wrong-password? pwd)
    (not (eq? pwd password)))
  (define (dispatch pwd m)
    (cond ((wrong-password? pwd) handle-incorrect-password)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)
```

We create an additional procedure `make-joint` that uses the account if the password matches. Otherwise it signals incorrect password.

```scheme
(define (make-joint account account-password joint-password)
  (define (correct-password? pwd)
    (eq? pwd joint-password))
  (lambda (input-pwd request)
    (if (correct-password? input-pwd)
        (account account-password request)
        (lambda (_) "Incorrect password"))))
```

Now we can test our procedure by creating a joint account and observing how the first account changes as we use the joint account.

```scheme
(define peter-acc (make-account 100 'open-sesame))
(define paul-acc
  (make-joint peter-acc 'open-sesame 'rosebud))
```

```
;; test linked account
(= ((peter-acc 'open-sesame 'withdraw) 40) 60)
(= ((paul-acc 'rosebud 'deposit) 40) 100)

;; test wrong password for Paul
(string=? ((paul-acc 'open-sesame 'withdraw) 100) "Incorrect password")

;; test insufficient funds for Peter
(= ((paul-acc 'rosebud 'withdraw) 100) 0)
(string=? ((peter-acc 'open-sesame 'withdraw) 1) "Insufficient funds")
```

**Exercise 3.8**

> Define a simple procedure `f` such that evaluating `(+ (f 0) (f 1))` will return 0 if the arguments to `+` are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

We let `f` initialize a local state variable `state` to 0. Then we construct `f` such that it returns a function `g` that always mutates `state` to the value of its argument $x$. The return value of `g` will be the old `state` value before the update.

```
(define f
  (let ((state 0))
    (define (g x)
      (let ((old-state state))
        (begin (set! state x)
               old-state)))
    g))
```

In the expression `(+ (f 0) (f 1))` if the arguments are evaluated left to right, then the first call to `f` will be `(f 0)` so `state` will equal to 0 and return value will be 0 since that was the old state value due to initialization. The subsequent `(f 1)` will update `state` to 1 but the function returns the old state value 0. The final value is thus `(+ 0 0) = 0`.

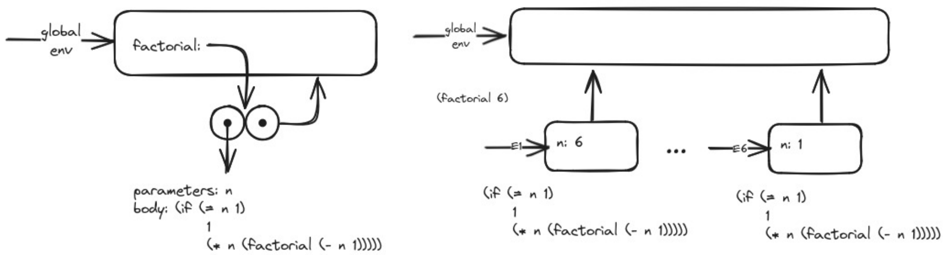
Consequently, if the arguments are evaluated right to left, then the first call to `f` will be `(f 1)` and `state` would be set to 1, but due to initialization of `state` the old state value is 0 which is what we output. The next call `(f 0)` would set the `state` to 0, however this time the old `state` value is 1 which is what we output. Hence we have `(+ 0 1) = 1` as desired.

## 1.2 The Environment Model of Evaluation

### Exercise 3.9

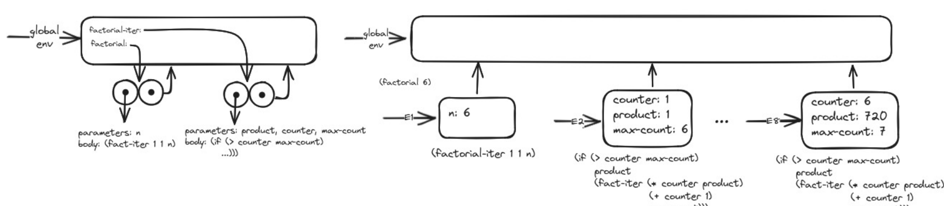
> Show the environment structures created by evaluating `(factorial 6)` using each version of the factorial procedure.

We start with the recursive version of the factorial program. `(factorial n)` is defined in the global environment which means that we must bind the name `factorial` to it there (and point back to the global environment). When `(factorial 6)` is invoked we create 6 different environments $E_1, \ldots, E_6$ where we have bound the formal parameter $n$.




In the last environment $E_6$ the function will return 1 since $n = 1$. This returned value will be propagated back to the caller in $E_5$ which will use that value for its calculation. This will continue until we reach $E_1$ where the final result is returned.

Let's now look at the iterative version of the factorial program. Again `factorial` is defined in the global environment as is `factorial-iter`. When `(factorial 6)` is called, this time the first environment $E_1$ will need to lookup `(factorial-iter)` which is found the global environment. So we create a new environment $E_2$ where we bind the formal parameters `product`, `counter` and `max-count` using values from $E_1$. We create another frame $E_3$ for evaluating `factorial-iter` with parameters set by $E_2$. We iterate this way until we are in an environment where `count` greater than `max-count`. This happens in $E_8$ and then we return the value of `product` back to all previous callers.



**Exercise 3.10**

> Use the environment model to analyze this alternate version (using `let` expression) 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))
> ```

The alternative version of `make-withdraw` is defined as follows in the book.

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

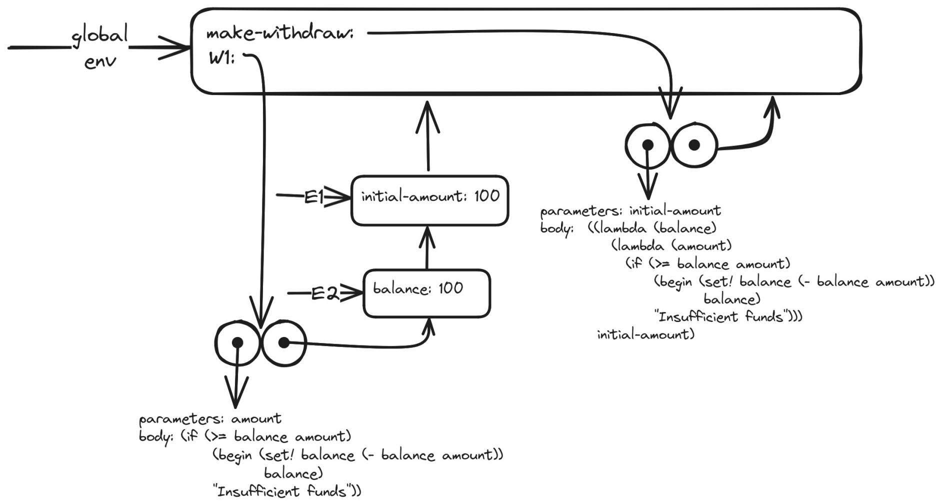
We rewrite it by recalling that `(let ((⟨var⟩ ⟨exp⟩)) ⟨body⟩)` syntactic sugar for `((lambda (⟨var⟩) ⟨body⟩) ⟨exp⟩)`.

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

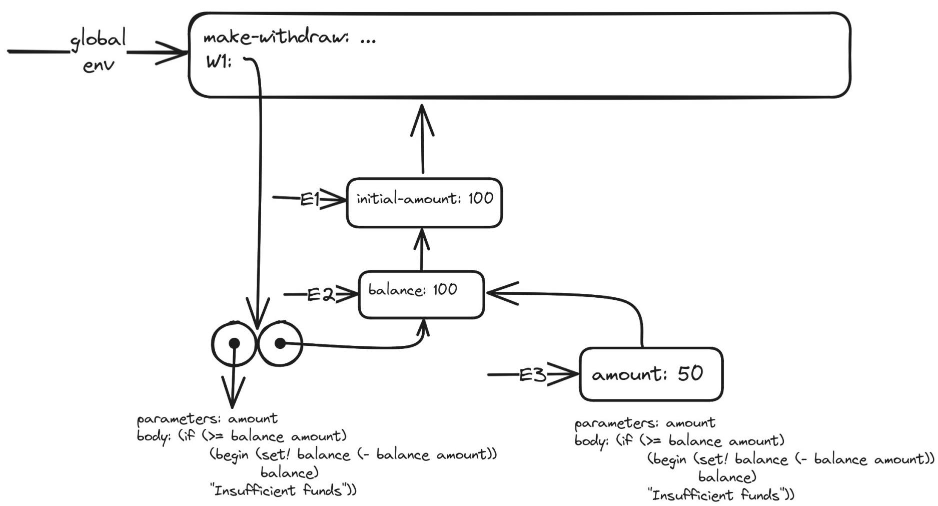
When `(define W1 (make-withdraw 100))` we first need to evaluate the sub-expression `(make-withdraw 100)`. To do that we create an environment $E_1$ where `initial-amount` is bound to 100 and evaluate the following expression.

```
((lambda (balance)
   (lambda (amount)
     (if (>= balance amount)
         (begin (set! balance (- balance amount)) balance)
       "Insufficient funds")))
 initial-amount) ;; bound to 100 in E_0
```

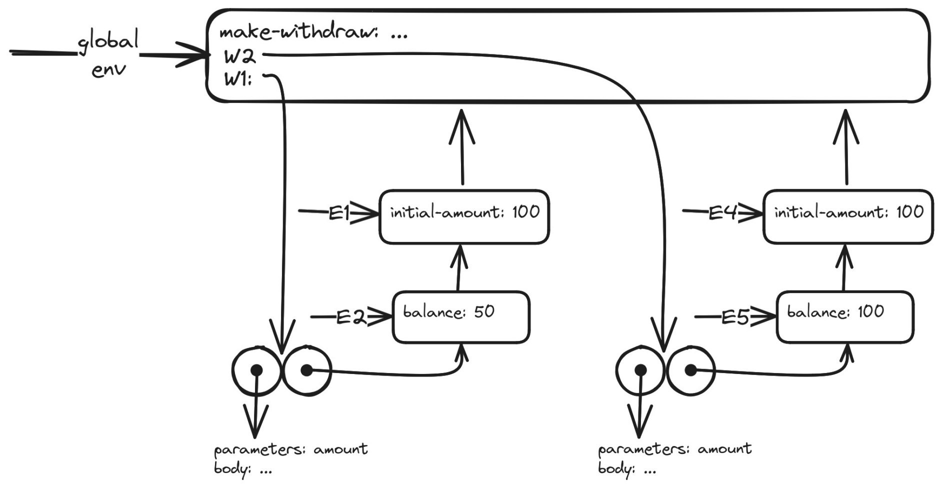
The result is a $\lambda$-expression together with an environment $E_2$ where `balance` is bound to 100. Since this new $\lambda$-expression was evaluated in $E_1$ it will point back to it rather than the global environment.




When (W1 50) is evaluated we create a new environment $E_3$ where amount is bound to 50. We then evaluate the body of the $\lambda$-expression in $E_3$ and lookup balance which we find in $E_2$. The result is 50 and the effect of set! is to change the value of balance in $E_2$ to 50. After this call is finished $E_3$ is discarded.



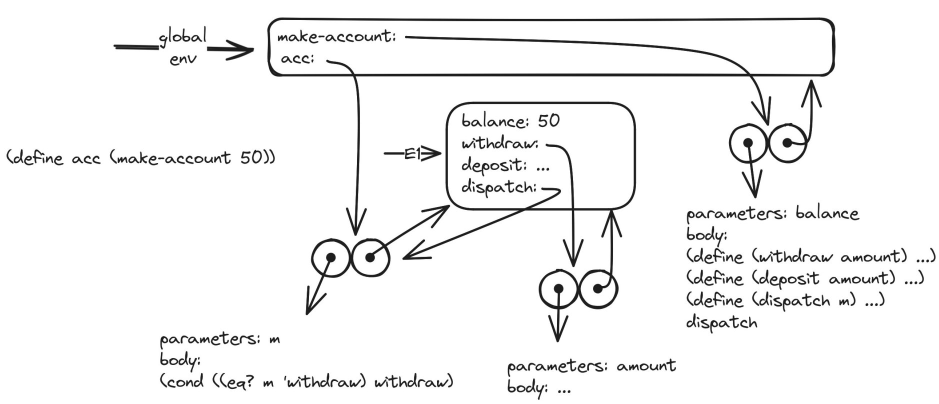

Now when we run (define W2 (make-withdraw 100)) we create a new environment $E_5$ where initial-amount is bound to 100. The new object W2 is evaluated within $E_5$ so its environment where balance is bound to 100 will point to it.


global
env
make-withdraw: ...
W2
W1:
E1
initial-amount: 100
E2
balance: 50
parameters: amount
body: ...
E4
initial-amount: 100
E5
balance: 100
parameters: amount
body: ...

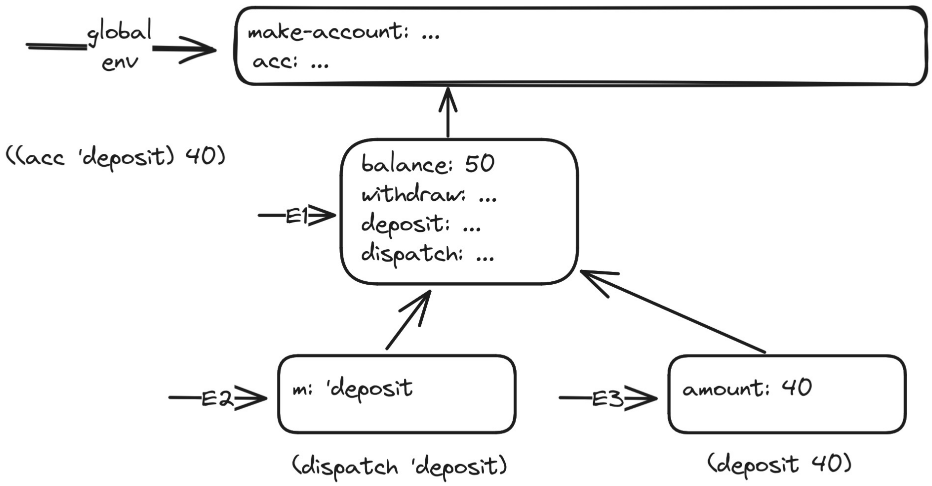
**Exercise 3.11**

> Consider the bank account procedure of `Section 3.1.1`. 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`?

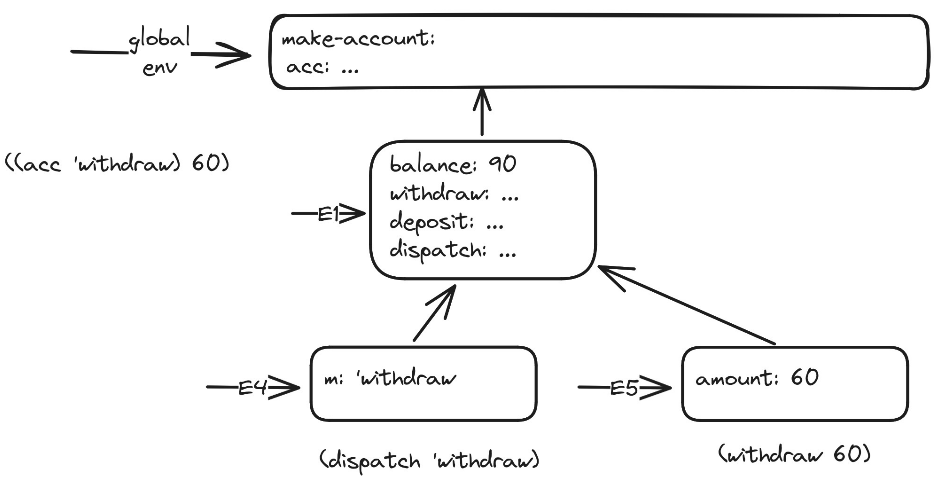
When `(define acc (make-account 50))` is called in the global environment we need to evaluate any sub-expressions. Beginning with the arguments `(make-account 50)`. Following the environment model we create a new frame in environment `E1` binding the formal paramter `balance` to 50. Then we bind all internal definitions of `withdraw`, `deposit` and `dispatch` in `E1`. Since `(make-account 50)` was called in the global environment `E1` will point to it. Since `(make-account 50)` returns `dispatch` that is what `acc` will be bound to in the global environment (which is where the `define` was called in first place).



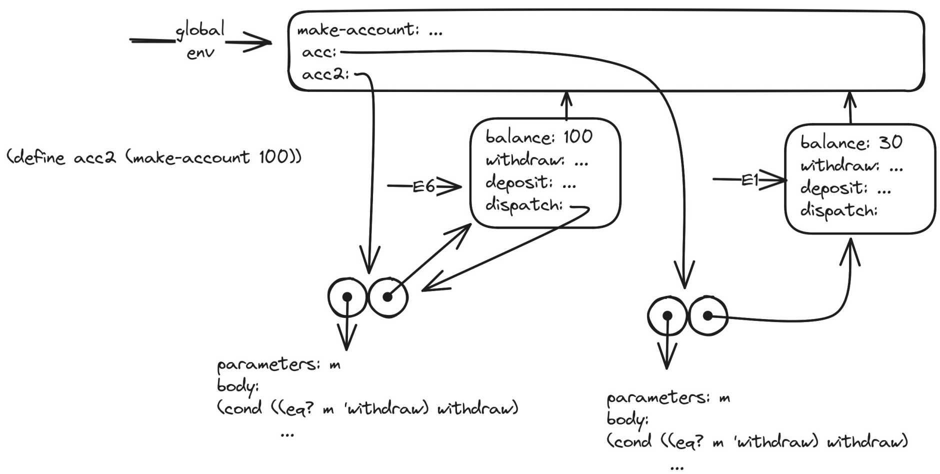

Now we proceed with `((acc ’deposit) 40)` by evaluating the sub-expression `(acc ’deposit)` in the global environment. `acc` points to the computational object `dispatch` and so we create a new environment `E2` where we bound the formal parameters of `dispatch` to `’deposit`. `E2` is enclosed by `E1` since that is the environment part of `dispatch`. The call to `(dispatch ’deposit)` returns the computational object `deposit`. Hence we need to evaluate `(deposit 40)`. We therefore create a new environment `E3` which will be enclosed by `E1`, again due to the fact that that is the environment part of `deposit`. In `E3` we bind the formal paramter of `deposit`, which is `amount` to 40. This call to `deposit` has the side-effect due to the use of `set!` in its body that mutates `balance` inside `E1`.




The `((acc 'withdraw) 60)` call is evaluated in a similar fashion creating two new ephermal environments `E4` and `E5`. The only difference is that `balance` is decremented.




Finally, the call `(define acc2 (make-account 100))` sets up a new environment `E6` where `balance` is bound to 100. We see here that the two accounts are kept distinct by the fact that they have different environments. The only environment structure that shared between the two accounts is the global environment.

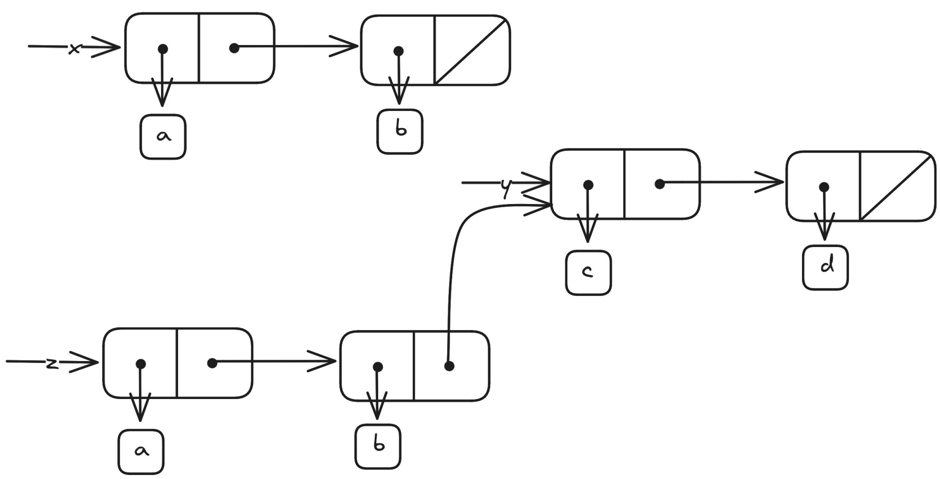

global env
make-account: ...
acc:
acc2:
(define acc2 (make-account 100))
balance: 100
withdraw: ...
deposit: ...
dispatch:
E6
balance: 30
withdraw: ...
deposit: ...
dispatch:
E1
parameters: m
body:
(cond ((eq? m 'withdraw) withdraw)
...
parameters: m
body:
(cond ((eq? m 'withdraw) withdraw)
...

## 1.3 Modeling with Mutable Data

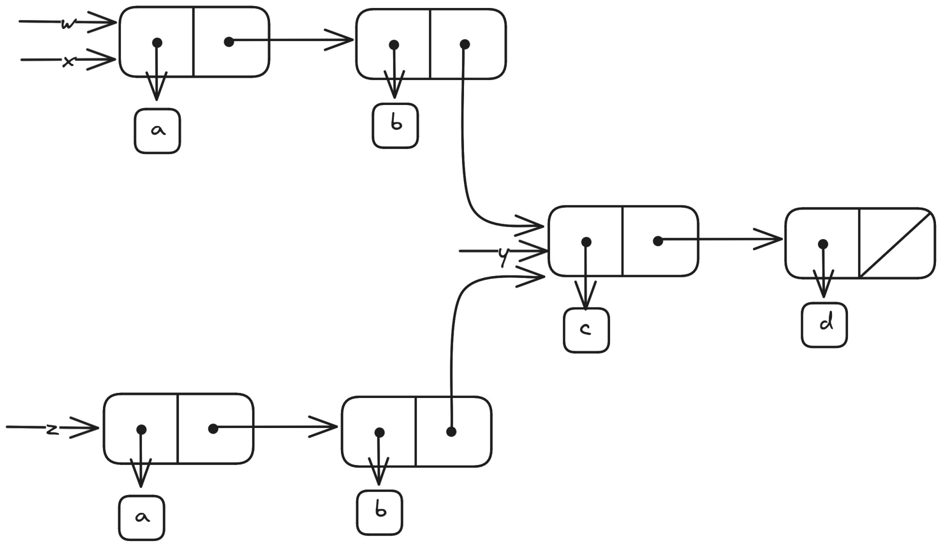
### Exercise 3.12

> Consider the interaction
>
> ```scheme
> (define x (list 'a 'b))
> (define y (list 'c 'd))
> (define z (append x y))
> z
> (a b c d)
> (cdr x)
> ⟨response⟩
> (define w (append! x y))
> w
> (a b c d)
> (cdr x)
> ⟨response⟩
> ```
>
> What are the missing `⟨response⟩`? Draw box-and-pointer diagrams to explain your answer.

We draw the box-and-pointer diagram for state up until defining the variable `z`.




In doing so we see that the the first missing `⟨response⟩` is `(b)`. Let's now draw the diagram after `w` is defined. We are using the mutator procedure `append!` so the list structure is modified in-place.

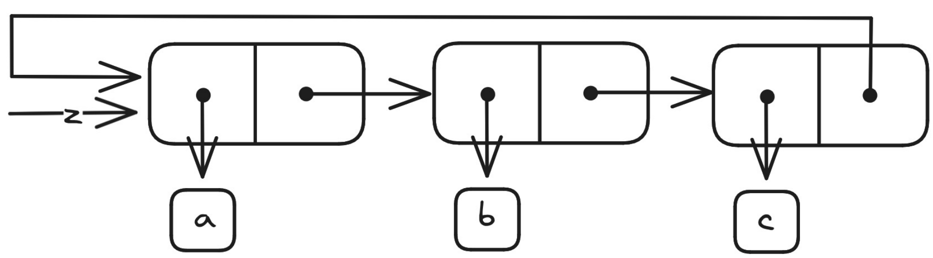



This time the missing response will be `(b c d)` since `x` was mutated when `w` was defined.

**Exercise 3.13**

> Consider the following make-cycle procedure, which uses the `last-pair` procedure defined in Exercise 3.12:
>
> ```
> (define (make-cycle x)
>   (set-cdr! (last-pair x) x)
>   x)
> ```
>
> Draw a box-and-pointer diagram that shows the structure `z` created by
>
> ```
> (define z (make-cycle (list 'a 'b 'c)))
> ```
>
> What happens if we try to compute `(last-pair z)`?

The procedure `make-cycle` will set the last pair's `cdr` of `(list 'a 'b 'c)` to the head of itself. This means that we have created a cycle over the `'(a b c)`. We draw the box-and-pointer diagram for `z` below.



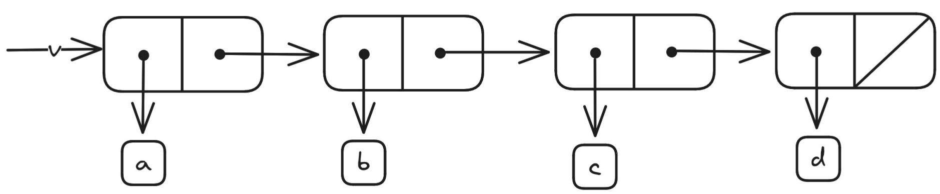

As can be seen in the diagram above, if we try to evaluate `(last-pair z)` we will get an infinite loop since the last pair of `z` points to itself.

**Exercise 3.14**

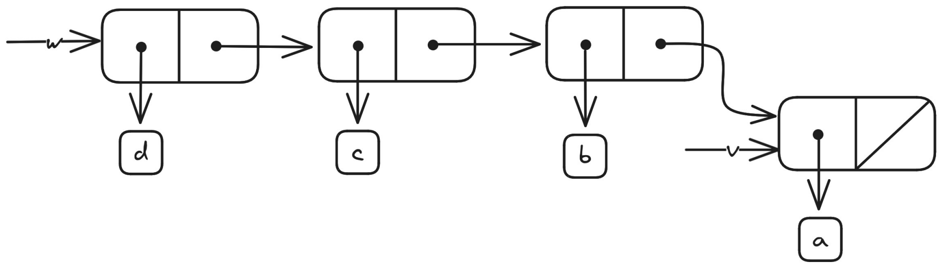
> The following procedure is quite useful, although obscure:
>
> ```
> (define (mystery x)
>   (define (loop x y)
>     (if (null? x)
>         y
>         (let ((temp (cdr x)))
>           (set-cdr! x y)
>           (loop temp x))))
>   (loop x '()))
> ```
>
> Draw the box-and-pointer diagram that represents the list to which v is bound. Suppose that we now evaluate `(define w (mystery v))`. Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating this expression. What would be printed as the values of `v` and `w`?

We can observe that the value of `x` in `loop` will always be the tail of the previous `x` value. So that `temp` would be `(a b c d)`, `(b c d)`, . . . , `(d)`, `()`. The `y` is always the previous iteration's mutated `x` value beginning with the empty list `()`. Since `x` is changed by taking its head and setting its `cdr` to be `y`, the values `y` take in each iteration are `()`, `(a)`, `(b a)`, . . . , `(d c b a)`. This means that `(mystery x)` is effectively reversing the list `x`.

We draw the box-and-pointer diagram for `v` below.



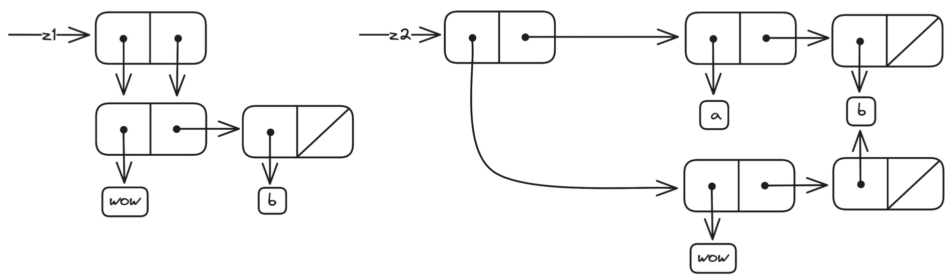

After the call `(define w (mystery v))` we have reversed the list `v` and bound it to `w`. However, the first `set-cdr!` in `loop` will mutate the value pointed by `v` by setting the `cdr` to `y` which is initialized by the empty list. In the next iteration, `v` will be passed in `loop` as `y` which is not mutated and in fact dropped. This means we only mutate the value of `v` once to `(a)`. We draw the box-and-pointer diagram for `v` and `w` below.


w
d
c
b
v
a

**Exercise 3.15**

> Draw box-and-pointer diagrams to explain the effect of set-to-wow! on the structures `z1` and `z2` above.

We draw the box-and-pointer diagram for `z1` and `z2` after the effects calling `set-to-wow!` on both of them, respectively.



### Exercise 3.16

> The number of pairs in any structure is the number in the car plus the number in the `cdr` plus one more to count the current pair.
>
> ```
> (define (count-pairs x)
>   (if (not (pair? x))
>       0
>       (+ (count-pairs (car x))
>          (count-pairs (cdr x))
>          1)))
> ```
>
> Show that this procedure is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's procedure would return 3; return 4; return 7; never return at all.

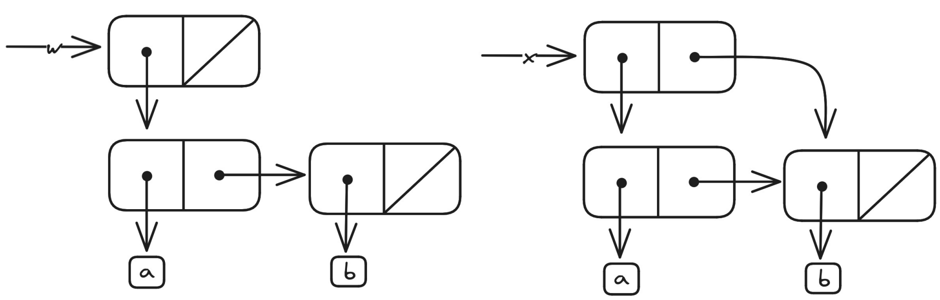
We begin with the first two cases, where we construct `w` to return 3 and `x` to return 4.

```
(define b (cons 'b '()))
(define a (cons 'a b))

(define w (cons a '()))
(count-pairs w)                         ; returns 3

(define x (cons a b))
(count-pairs x)                         ; returns 4
```

The diagrams for these two cases are shown below.



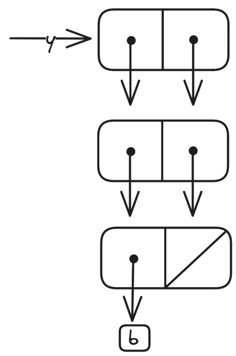

For the third we construct `y` to return 7.

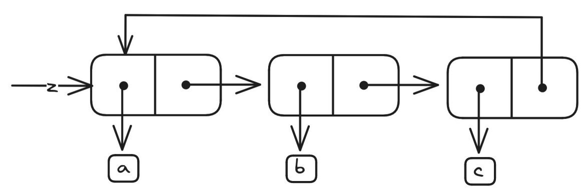
```
(define bb (cons b b))
(define y (cons bb bb))
(count-pairs y)                         ; returns 7
```

For the last case we simply let `z` to be a cycle using the `make-cycle` procedure from `Exercise 3.13`. We show diagrams below

```scheme
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)

(define z (make-cycle (cons 'a (cons 'b (cons 'c '())))))
;; (count-pairs z)                          ; will never halt
```



**Exercise 3.17**

> Devise a correct version of the count-pairs procedure of `Exercise 3.16` that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

We will implement a set data structure, admittedly ineffienct. We will use the `memq` procedure to check if an element is in the set and `cons` to add the element to the set. This works because `memq` uses `eq?` under the hood which checks equality by pointers. The number of pairs will be the length of our set.

```
(define (count-pairs x)
  (define seen-pair '())
  (define (count x)
    (cond ((not (pair? x)) 0)
          ((memq x seen-pair) 0)
          (else (set! seen-pair (cons x seen-pair))
                (+ (count (car x))
                   (count (cdr x))
                   1))))
  (count x))
```

Using this new version of `count-pairs` with the list structres we defined in `Exercise 3.16`, all results are returned correctly as 3.

### Exercise 3.18

> Write a procedure that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by taking successive `cdrs` would go into an infinite loop. `Exercise 3.13` constructed such lists.

We can construct a path of `cons` by following each successive `cdr` of a list. If such a path of `cons` has a cycle, then at least one of the `cons` in the path points back to a previous `cons` in the path. We write the procedure `has-cycle?` using this fact.

```
(define (has-cycle? x)
  (define (cycle-in-path? x path)
    (cond ((not (pair? x)) #f)
          ((memq (cdr x) path) #t)
          (else (cycle-in-path? (cdr x) (cons x path)))))
  (cycle-in-path? x (list x)))
```

Let's test this procedure by creating a cycle and a non-cycle. From `Exercise 3.13` we use the procedure `make-cycle`.

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)
```

Next let us define `x` with no cycle and `z` with a cycle.

```
(define x '(a b c))
(define z (cons 'a (make-cycle (cons 'a 'b))))
```

We can now test our procedure by calling `has-cycle?` on `x` and `z`.

```
(has-cycle? x)                          ; returns #f
(has-cycle? z)                          ; returns #t
```

However, note that we construct the path by `cdr`-ing down the list. This means that if we have a cycle in the `car` part of the list, then we will not detect it. We could easily amend this by adjusting the `cycle-in-path?` to check for cycles in the `car` part of the list as well.

```
(define (cycle-in-path? x path)
  (cond ((not (pair? x)) #f)
        ((or (memq (car x) path) (memq (cdr x) path)) #t)
        (else (or (cycle-in-path? (car x) (cons x path))
                  (cycle-in-path? (cdr x) (cons x path))))))
```

### Exercise 3.19

> Redo `Exercise 3.18` using an algorithm that takes only a constant amount of space. (This requires a very clever idea.)

We can use the famous `tortoise-and-hare` algorithm for detecting cycles in a list. The idea is to have two pointers, one that moves one step at a time and another that moves two steps at a time. If there is a cycle in the list, then the two pointers will eventually point to the same `cons` in the list. We omit the proof here.

Since we only use two pointers to travers in addition to the list, this algorithm takes constant space. We write the procedure `has-cycle?` using this idea.

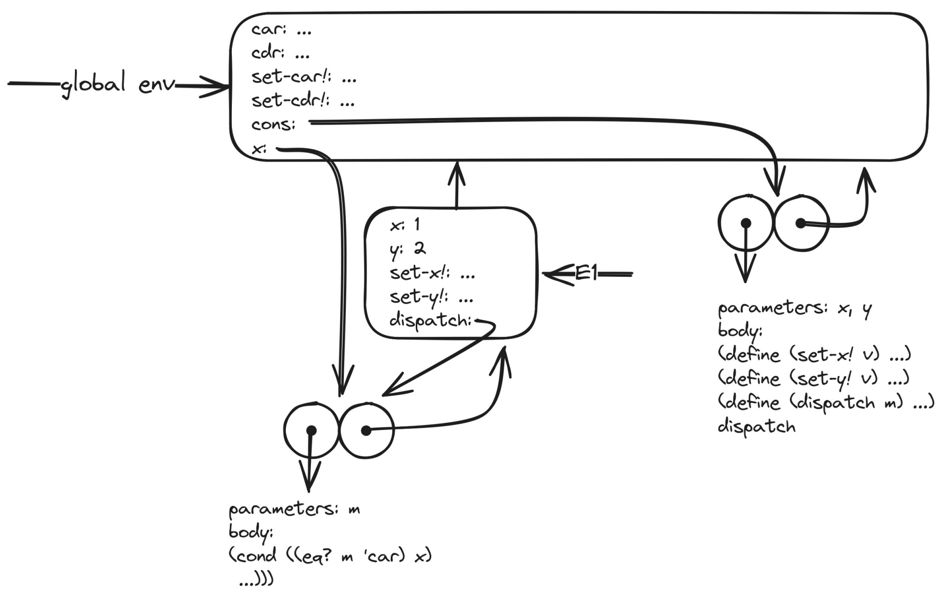
```
(define (has-cycle? x)
  (define (tortoise-and-hare tortoise hare)
    (cond ((not (pair? hare)) #f)
          ((eq? tortoise hare) #t)
          (else (tortoise-and-hare (cdr tortoise) (cddr hare)))))
  (tortoise-and-hare x (cdr x)))
```

Note that the procedure above only works for non-empty lists. We can easily amend this by adding a conditional before the call to `tortoise-and-hare` in `has-cycle?`. But we omit it to not clutter the code.

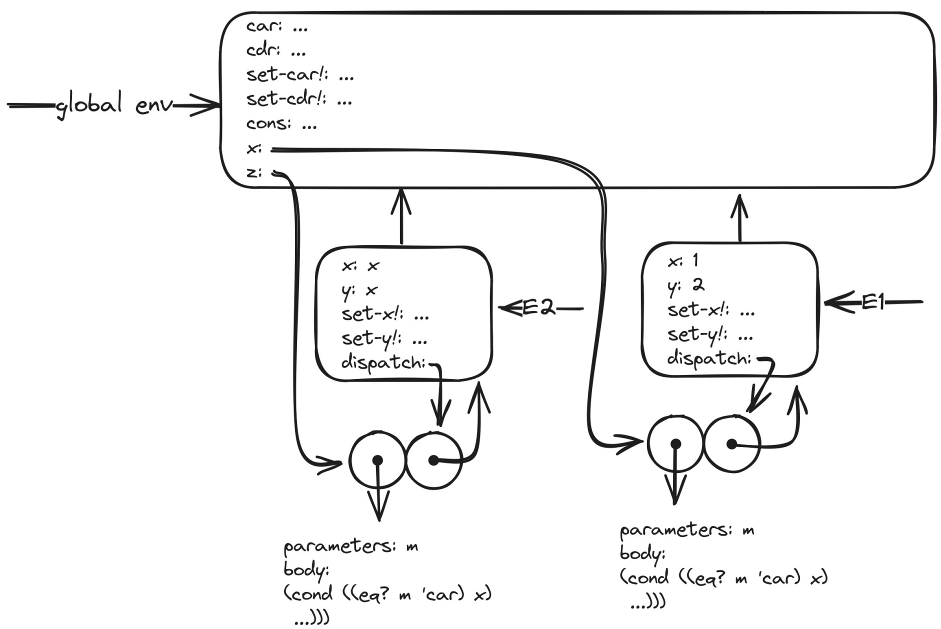
**Exercise 3.20**

> Draw environment diagrams to illustrate the evaluation of the sequence of expressions
>
> ```
> (define x (cons 1 2))
> (define z (cons x x))
> (set-car! (cdr z) 17)
> (car x)
> 17
> ```
>
> using the procedural implementation of pairs given above. (Compare `Exercise 3.11`.)

`cons` points to a procedure enclosed by the global environment with parameters `x, y`. When defining `x` we create a new frame `E1` where we bound `cons` parameters to `1, 2` and evaluate the body. We see the result in the diagram below.




Similarly, when defining `z` we create a new frame `E2`, but this time bound `cons` parameters to `x, x`.

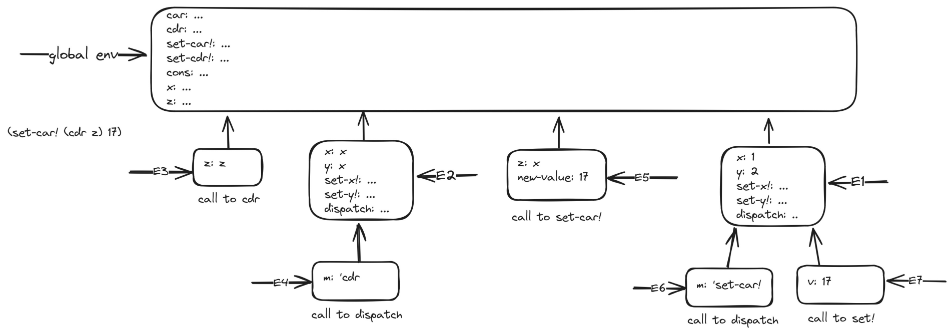



Now we want to evaluate the compound expression `(set-car! (cdr z) 17)`. First we need to evaluate all sub-expressions. This means we begin with `(cdr z)` since `17` is a primitve. Therefore we create a new frame `E3` where we bound `cdr` parameters to `z` and evaluate the body.

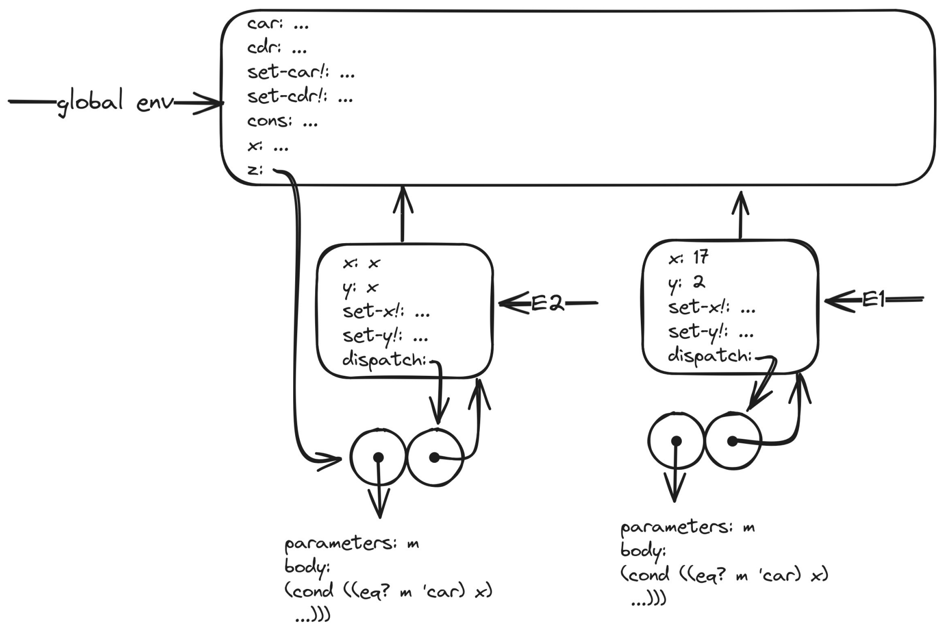
This leaves us the expression `(z 'cdr)` and we look in the global enviroment after `z`. We find that `z` points to a procedural object called `dispatch` in `E2`. As before, create a new frame `E4` and bind the `dispatch`'s parameter `m` to `'cdr`. Evaluation of the body gives us the symbol `x`.

The expression we need to evaluate now is `(set-car! x 17)`. At this point every sub-expression is known in the global environment so we must apply `set-car!` to the parameters. Hence, we create a new frame `E5` and bind `set-car!`'s parameters to `x, 17` and leaves us with the expression `(((x 'set-car!) 17) x)`.

The sub-expression `(x 'set-car!)` is evaluated in a new frame `E6` and results in `set-x!` procedure defined in enviroment `E1`. We have `((set-x! 17) x)` left to evaluate. For `(set-x! 17)` we create a new frame `E7` enclosed by `E1`. Following the procedure `set-x!` and binding its parameter to `17` has the effect of setting the `car` of `x` to the value `17`. Lastly, we just return `x` back to the caller.




The final result is that the `car` value of `x` was mutated. Since `cdr z` points to `x`, `z` will also be affected by this as a side-effect.




`Exercise 3.11` is similar in that we have internal definitions and variables. So the two evaluations of `cons` leads to two enviroments `E1` and `E2`, both of which have their own `dispatch`, exactly as in `Exercise 3.11`. However, a difference here is that when defining `z` we bind it to the previously created `x` – in effect coupling them as we've seen in the final result.

**Exercise 3.21**

> Ben Bitdiddle decides to test the queue implementation described above. “It’s all wrong!” he complains. “The interpreter’s response shows that the last item is inserted into the queue twice. Show why Ben’s examples produce the printed results that they do. Define a procedure `print-queue` that takes a queue as input and prints the sequence of items in the queue.

A sequence of `cons` terminated by `'()` is printed as a list in Scheme. Hence

```
(cons 'a '())
(a)
(cons 'a 'b)
(a . b)
```

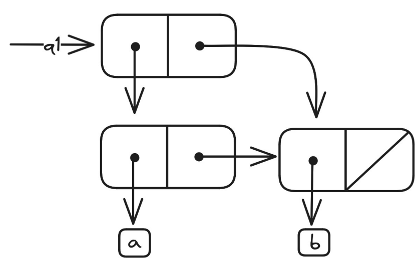
When we evaluate the following expressions, as Ben Bitdiddle does,

```
(define q1 (make-queue))
(insert-queue! q1 'a)
```

then `q1` will point to the pair `(cons (cons 'a '()) (cons 'a '()))`. Therefore the print will be

```
((a) a)
```

There’s a nested parenthesis to distinguish that the `car` is a list in its own, which we wouldn’t have seen if it was printed without them like so `(a a)`. Subsequent insertion `(insert-queue! q1 'b)` will make `q1` to point to the pair `(cons (cons 'a (cons 'b '())) (cons 'b '()))` and print `((a b) b)`. However, do note that `rear-ptr` is just a pointer to the `last-pair` in `(cons 'a (cons 'b '()))` as can be seen in the diagram below.




To define a procedure `print-queue` we simply traverse the queue starting from the `front-ptr` and `cdr` until we reach the end of the queue. We print the `car` of each pair along the way. But this is what Scheme already does when we print a list. So we can just return the `front-ptr` and Scheme will print it for us.

```
(define (print-queue q)
  (front-ptr q))
```

### Exercise 3.22

> Complete the definition of `make-queue` and provide implementations of the queue operations using this representation.
>
> ```
> (define (make-queue)
>   (let ((front-ptr . . . )
>         (rear-ptr . . . ))
>     ⟨definitions of internal procedures⟩
>     (define (dispatch m) . . .)
>     dispatch))
> ```

We initialize `front-ptr` and `rear-ptr` to be empty lists. Whenever we insert an `item`, we simply tack the pair `(cons item '())` onto the current rear, and then `set!` the `rear-ptr` to this pair. For deletion we simply set `front-ptr` to its `cdr`.

```
(define (make-queue)
  (let ((front-ptr '())
        (rear-ptr '()))
    (define (empty?) (null? front-ptr))
    (define (insert! item)
      (let ((new-pair (cons item '())))
        (cond ((empty?)
               (set! front-ptr new-pair)
               (set! rear-ptr new-pair)
               front-ptr)
              (else
               (set-cdr! rear-ptr new-pair)
               (set! rear-ptr new-pair)
               front-ptr))))
    (define (delete!) (set! front-ptr (cdr front-ptr)) front-ptr)
    (define (dispatch m)
      (cond ((eq? m 'empty?) (empty?))
            ((eq? m 'front) (car front-ptr))
            ((eq? m 'insert!) insert!)
            ((eq? m 'delete) (delete!))
            ((eq? m 'print) front-ptr)))
    dispatch))
```

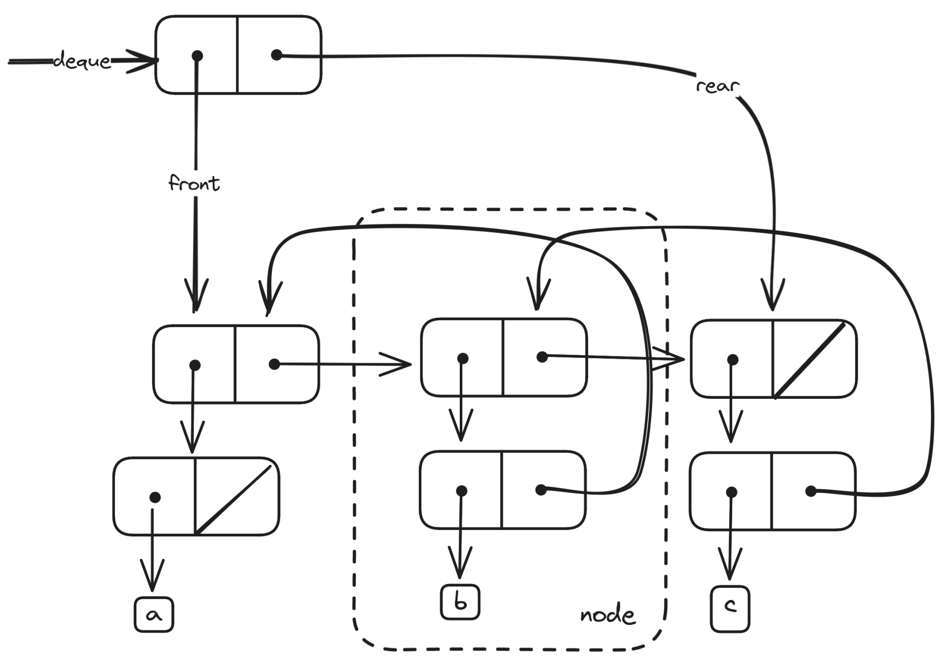
Now we define the rest of interface for `queue` in terms of the `dispatch`.

```
(define (delete-queue! queue) (queue 'delete))
(define (empty-queue? queue) (queue 'empty?))
(define (front-queue queue) (queue 'front))
(define (insert-queue! queue item) ((queue 'insert!) item))
(define (print-queue queue) (queue 'print))
```

**Exercise 3.23**

> A *deque* ("double-ended queue") is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make-deque`, the predicate `empty-deque?`, selectors `front-deque` and `rear-deque`, mutators `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, and `rear-delete-deque!`. Show how to represent deques using pairs, and give implementations of the operations. All operations should be accomplished in $\Theta(1)$ steps.

We begin by making an abstraction of the elements that go into our deque. This is not necessary but helps with readiblity of the code. Each element is called a `node` and consists of an item and two pointers `prev` and `next`. If a `node` is the first element in the deque, then its `prev` pointer will be empty. Similarly, if a `node` is the last element in the deque, then its `next` pointer will be empty. We can define the `deque` simply by a `cons` with elements of type `node`. The image below shows a `deque` of three nodes which hold items `a, b, c`, respectively. We mark the middle `node` in the image with a dashed box.




As can be seen in the diagram above, we define a `node` to consts of two pairs. In addtion, we provide helpers `prev`, `next` and their mutators to facilitate creating the `deque` later on.

```
(define (make-node item)
  (cons (cons item '()) '()))
```

```scheme
(define (item node) (caar node))
(define (prev node) (cdar node))
(define (next node) (cdr node))
(define (set-prev! node item) (set-cdr! (car node) item))
(define (set-next! node item) (set-cdr! node item))
```

We can now define the deque and implements its interface with help of the facilities of node.

```scheme
(define (make-deque) (cons '() '()))
(define (front-deque deque) (car deque))
(define (rear-deque deque) (cdr deque))
(define (empty-deque? deque) (null? (front-deque deque)))
(define (set-front! deque item) (set-car! deque item))
(define (set-rear! deque item) (set-cdr! deque item))
(define (front-insert-deque! deque item)
  (cond ((empty-deque? deque)
         (let ((new-pair (make-node item)))
           (set-front! deque new-pair)
           (set-rear! deque new-pair)
           #t))
        (else
         (let ((new-item (make-node item)))
           (set-prev! (front-deque deque) new-item)
           (set-next! new-item (front-deque deque))
           (set-front! deque new-item)
           #t))))

(define (rear-insert-deque! deque item)
  (cond ((empty-deque? deque)
         (let ((new-pair (make-node item)))
           (set-front! deque new-pair)
           (set-rear! deque new-pair)
           deque))
        (else
         (let ((new-rear (make-node item))
               (current-rear (rear-deque deque)))
           (set-next! current-rear new-rear)
           (set-prev! new-rear current-rear)
           (set-rear! deque new-rear)
           #t))))

(define (front-delete-deque! deque)
  (cond ((empty-deque? deque) #t)
        ((null? (next (front-deque deque)))
         (set-front! deque '())
         (set-front! deque '())
         #t)
        (else
         (let ((new-front (next (front-deque deque))))
           (set-prev! new-front '())
```

```scheme
          (set-front! deque new-front)
          #t))))

(define (rear-delete-deque! deque)
  (cond ((empty-deque? deque) #t)
        ((null? (prev (rear-deque deque)))
         (set-rear! deque '())
         (set-front! deque '())
         #t)
        (else
         (let ((new-rear (prev (rear-deque deque))))
           (set-next! new-rear '())
           (set-rear! deque new-rear)
           #t))))
```

In order to avoid Scheme printing the cycle we have in the `deque`, some of our operations simply output `#t`. However, we also provide the `print-deque` procedure which prints the `item` of each `node` in the `deque`.

```scheme
(define (print-deque deque)
  (define (iter node)
    (if (null? node)
        '()
        (cons (item node) (iter (next node)))))
  (iter (front-deque deque)))
```

Let's test our implementation by creating a `deque` and inserting and deleting elements.

```scheme
(define d (make-deque))
(empty-deque? d)                        ; returns #t
(front-insert-deque! d 2)
(empty-deque? d)                        ; returns #f
(front-insert-deque! d 1)
(rear-insert-deque! d 3)
(print-deque d)                         ; returns (1 2 3)

(rear-delete-deque! d)
(print-deque d)                         ; returns (1 2)
(front-delete-deque! d)
(print-deque d)                         ; returns (2)
(rear-delete-deque! d)
(empty-deque? d)                        ; returns #t
```

### Exercise 3.24

> Design a table constructor `make-table` that takes as an argument a `same-key?` procedure that will be used to test "equality" of keys. `make-table` should return a `dispatch` procedure that can be used to access appropriate `lookup` and `insert!` procedures for a local table.

The only difference would be that our `assoc` in this case uses `same-key?` to test for equality.

```
(define (assoc key records)
  (cond ((null? records) false)
        ((same-key? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

This new `assoc` can be used as a local definition inside the `make-table` procedure.

```
(define (make-table same-key?)
  (let ((local-table (list '*table*)))
    (define (assoc key records)
      (cond ((null? records) false)
            ((same-key? key (caar records)) (car records))
            (else (assoc key (cdr records)))))
    (define (lookup key)
      (let ((record (assoc key (cdr local-table))))
        (if record
            (cdr record)
            false)))
    (define (insert! key value)
      (let ((record (assoc key (cdr local-table))))
        (if record
            (set-cdr! record value)
            (set-cdr! local-table
                      (cons (cons key value)
                            (cdr local-table)))))
      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation: TABLE" m))))
    dispatch))
```

To test we simply need to provide a `same-key?` procedure and create a table with it

```
(define (same-key? key-1 key-2) (< (abs (- key-1 key-2)) 0.5))
(define operation-table (make-table same-key?))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```

and now we are ready to use the table.

```
(put 1.0 'a)
(put 2.0 'b)
(get 1.4)                                ; returns 'a
(get 2.5)                                ; returns #f
(get 1.9)                                ; returns 'b
```

**Exercise 3.25**

> Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The `lookup` and `insert!` procedures should take as input a list of keys used to access the table.

To generalize the table we need to have the ability to create sub-tables for any key where appropriate. We can do this by using the same `make-table` procedure as in `Exercise 3.24` but modify `insert!` to create sub-tables on the fly when needed. To do so we have to take care of a few base cases.

- If we only have one key `(null? (cdr keys))` we either have to insert the value in the record if it exists or put it inside a new record that we store the table.
- If we have more than one key we need to check if the record exists `(eq? record false)`.
  - If it does not exist we need to create a new sub-table and recursively insert the value in it.
  - If it does exist we need to check if it is a sub-table `(pair? (cdr record))`.
    - If it is a sub-table we need simply recursively insert the value in the sub-table.
    - If it is not a sub-table we remove the value-part of the record `(set-cdr! record '())` and insert our value in the record as if the record would be a sub-table `(insert! (cdr keys) value record))`.

```
(define (insert! keys value table)
  (let ((record (assoc (car keys) (cdr table))))
    (cond ((null? (cdr keys))
           (if record
               (set-cdr! record value)
               (set-cdr! table (cons (cons (car keys) value) (cdr table)))))
          (else
           (cond ((eq? record false)
                  (let ((new-table (cons (cons (car keys) '()) (cdr table))))
                    (set-cdr! table new-table)
                    (insert! (cdr keys) value (car new-table))))
                 ((not (pair? (cdr record)))
                  (set-cdr! record '())
                  (insert! (cdr keys) value record))
                 (else (insert! (cdr keys) value record)))))))
```

We also need to modify `lookup` to be able to traverse the sub-tables given a set of keys. This is fairly straight forward and we can use the same `assoc` as in `Exercise 3.24`.

```scheme
(define (lookup keys table)
  (let ((record (assoc (car keys) (cdr table))))
    (cond ((null? (cdr keys))
           (if record
               (cdr record)
               false))
          ((and record (pair? (cdr record)))
           (lookup (cdr keys) record))
          (else false))))
```

Now we can put it all together under `make-table` which our interface will use.

```scheme
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup keys table)
      (let ((record (assoc (car keys) (cdr table))))
        (cond ((null? (cdr keys))
               (if record
                   (cdr record)
                   false))
              ((and record (pair? (cdr record)))
               (lookup (cdr keys) record))
              (else false))))
    (define (insert! keys value table)
      (let ((record (assoc (car keys) (cdr table))))
        (cond ((null? (cdr keys))
               (if record
                   (set-cdr! record value)
                   (set-cdr! table (cons (cons (car keys) value) (cdr table)))))
              (else
               (cond ((eq? record false)
                      (let ((new-table (cons (cons (car keys) '()) (cdr table))))
                        (set-cdr! table new-table)
                        (insert! (cdr keys) value (car new-table))))
                     ((not (pair? (cdr record)))
                      (set-cdr! record '())
                      (insert! (cdr keys) value record))
                     (else (insert! (cdr keys) value record)))))))
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) (lambda (keys) (lookup keys local-table)))
            ((eq? m 'insert-proc!) (lambda (keys value) (insert! keys value
            ↪  local-table)))
            (else (error "Unknown operation: TABLE" m))))
    dispatch))
```

Let's make a table to test our implementation with

```scheme
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
(define print (operation-table 'print))
```

which we use in the test below

```
(put '(1) 'a)
(put '(2) 'b)
(get '(1))                          ; returns 'a
(get '(2))                          ; returns 'b
(put '(1 1) 'c)
(put '(1 2) 'd)
(get '(1))                          ; returns table ((1 . c) (2 . d))
(get '(1 1))                        ; returns 'c
(get '(1 2))                        ; returnc 'd
(get '(2))                          ; returns 'b
```

### Exercise 3.26

> Describe a table implementation where the (key, value) records are organized using a binary tree, as- suming that keys can be ordered in some way (e.g., numerically or alphabetically).

We describe an implementation of a one-dimensional table using binary trees. From `Chapter 2` in `Section 2.3.3` we have the interface for a binary tree

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= (car x) (car (entry set))) set)
        ((< (car x) (car (entry set)))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> (car x) (car (entry set)))
         (make-tree (entry set) (left-branch set)
                    (adjoin-set x (right-branch set))))))
```

where we made a slight modification to `adjoin-set` so that it works with key-value pairs. We also want a procedure that fetch us an entry if it exists. Such a procedure is easily gotten by modifying `element-of-set?` to return the actual entry it finds as opposed to a boolean. We call this procedure `find-entry` and adopt it so it works for key value pairs.

```
(define (find-entry key set)
  (cond ((null? set) false)
        ((= key (car (entry set))) (entry set))
        ((< key (car (entry set))) (find-entry key (left-branch set)))
        (else (find-entry key (right-branch set)))))
```

Now we use the binary tree to implement our table. We let an empty table be represented as the empty list `'()`.

```
(define (make-table)
  (let ((local-table '()))
    (define (lookup key)
      (let ((record (find-entry key local-table)))
        (if record
            (cdr record)
            false)))
    (define (insert! key value)
      (let ((record (find-entry key local-table)))
        (if record
```

```scheme
            (set-cdr! record value)
            (set! local-table (adjoin-set (cons key value) local-table))))
      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unkown operation: TABLE" m))))
    dispatch))
```

Note that `lookup` is nearly identical to `element-of-set?` procedure from `Chapter 2`. To conclude this exercise we create a test

```scheme
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))

(put 1 'a)
(put 2 'b)
(get 1)                             ; returns 'a
(get 2)                             ; returns 'b
(put 1 'c)
(get 1)                             ; returns 'c
(get 2)                             ; returns 'b
```

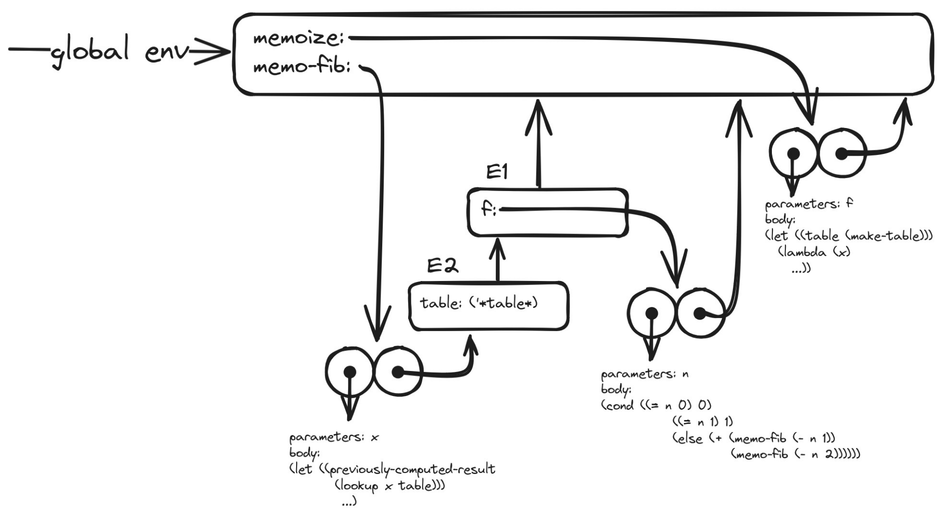
It is easy to see how we can extend this implementation to work non-numerical keys. We simply need to provide `same-key?`, `less-than?` and `greater-than?` procedures to `adjoin-set` and `find-entry`.

**Exercise 3.27**

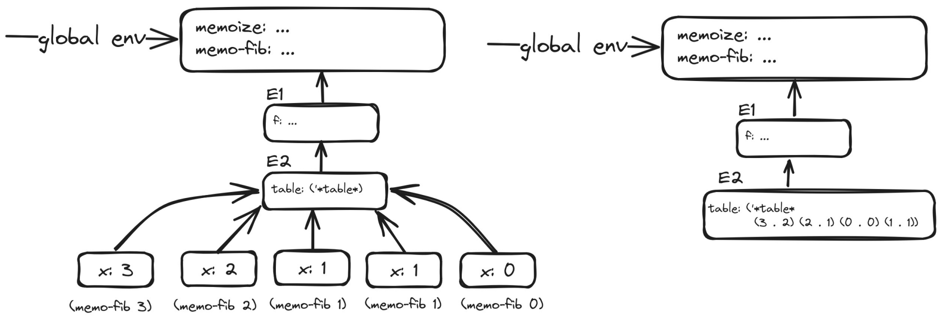
> Draw an environment diagram to analyze the computation of `(memo-fib 3)`. Explain why `memo-fib` computes the $n^{\text{th}}$ Fibonacci number in a number of steps proportional to $n$. Would the scheme still work if we had simply defined memo- fib to be `(memoize fib)`?

Beginning with the evaluation of `(define (memoize f) ...)` we get a procedural object which is enclosed by the global environment. This object has `f` as its formal parameter.

Next we evaluate the expression `(define memo-fib (memoize (lambda (n) ...)))`. First evaluate the sub-expression that is `(lambda (n) ...)`. This results in a procedural object with formal parameter `n` and since we evaluated this in the global environment that would enclose this object. Now we look up `memoize` in the global environment and bind its formal parameter `f` to the procedural object we just created in a new frame `E1`. This leads us to a `let` expression we can now evaluate in a new frame `E2` which would be enclosed by `E1`. In `E2` we bind `table` to the result of `make-table`. This results in a new procedural object which is enclosed by `E2` and has `x` as its formal parameter. This object is returned and bound to `memo-fib` in the global environment.

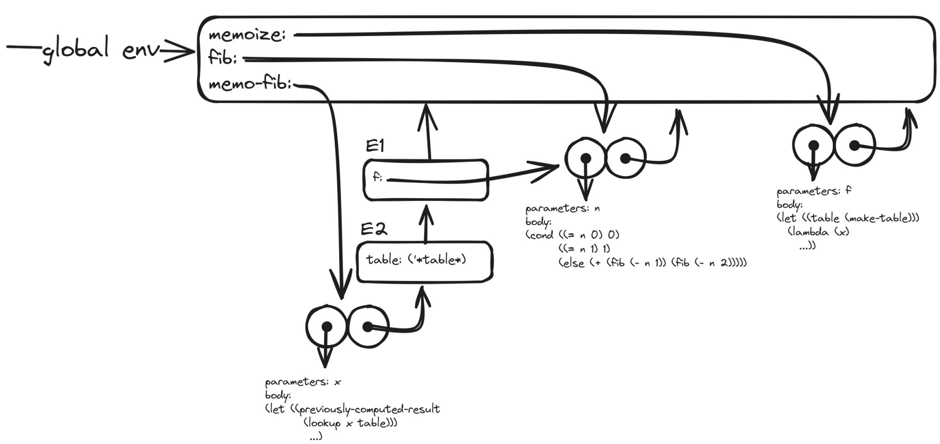



When we call `(memo-fib 3)` we create a new environment where we have bound `x` to 3. We run the lookup procedure and find that there is no entry for this argument. Hence we must use `f` to calculate the result which we can insert into the `table`. We find `f` in `E1` from `E2`. We create a new frame where `n` is bound to 3 and evaluate the body of `f`. This results in us having to call `(memo-fib 2)` and `(memo-fib 1)`. We find these in the global environment enclosed by `E2`, which means both of these call will have access to the same `table` variable. These in turn will have to be evaluated and can lookup/insert data into the `table` that can be accessed by later calls to `memo-fib`. At the end of the computation we have a `table` that contains the results of all calls to `memo-fib`. Here we have assumed that the right-most argument gets evaluated first. This will impact the order in which the `table` is filled.




We now show that the number of steps to calculate the $n^{\text{th}}$ Fibonacci number is proportional to $n$. We see that in each call to `memo-fib` we spawn two new calls to `memo-fib` with $n-1$ and $n-2$ as arguments. So the number of steps to get to `0` is proportional to $n$. However, note that the call `(memo-fib n-1)` would also call `(memo-fib n-2)` which means we would calculate `(memo-fib n-2)` at least twice. But since we store previous results in the `table` we can simply look up the result of `(memo-fib n-2)` in the `table` instead of recalculating it and thus avoid the extra computation. This is why `memo-fib` computes the $n^{\text{th}}$ Fibonacci number in a number of steps proportional to $n$.

Lastly, the scheme would work sub-optimally if we had `(define memo-fib (memoize fib))`. This is because only the result of the first call `(memo-fib 3)` would be inserted in the `table`. All of the other recursive calls to calculate the previous Fibonacci number use `fib`, which is defined in the global environment. `fib` is a procedure that does not store anything in a `table` (even if it would it has no access to `table` defined in `E2`).



### Exercise 3.28

> Define an `or-gate` as a primitive function box. Your `or-gate` constructor should be similar to `and-gate`.

The difference between the `and-gate` and our `or-gate` would be the amount of delay and the logical procedure that the function box uses to compute the output signal. We define `logical-or` in similar fashion as the `logical-and`

```
(define (logical-or s t)
  (cond ((= s 0) t)
        ((= t 0) s)
        (else 1)))
```

We will use this procedure to implement our primitive function box

```
(define (or-gate a1 a2 output)
  (define (or-action-procedure)
    (let ((new-value
           (logical-or (get-signal a1) (get-signal a2))))
      (after-delay
       or-gate-delay
       (lambda () (set-signal! output new-value)))))
  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure)
  'ok)
```

**Exercise 3.29**

> Another way to construct an `or-gate` is as a compound digital logic device, built from and-gates and inverters. Define a procedure `or-gate` that accomplishes this. What is the delay time of the or-gate in terms of and- gate-delay and inverter-delay?

Let's see what happens to the truth table if we invert the inputs to a logical-and.

| $p$ | $q$ | $\neg p$ | $\neg q$ | $\neg p \land \neg q$ |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |

This is almost what we want. If we had inverted the output of $\neg p \land \neg q$ we would have the truth table for a `logical-or`. Hence we can build our `or-gate` using one `and-gate` and three `inverter`.

```
(define (or-gate a1 a2 output)
  (let ((b (make-wire)) (c (make-wire)) (d (make-wire)))
    (inverter a1 b)
    (inverter a2 c)
    (and-gate b c d)
    (inverter d output)
    'ok))
```

Our implementation has three `inverter` and one `and-gate`. The delay time of the `or-gate` is the sum of the delay times would be

$$2I_d + A_d,$$

where $I_d$ is the delay time of the `inverter` and $A_d$ is the delay of the `and-gate`. We only get twice the `inverter` delay because the first two inverters for wires $a_1, a_2$ run in parallel.

### Exercise 3.30

> Write a procedure `ripple-carry-adder` that generates this circuit [shown in Fig 3.27]. The procedure should take as arguments three lists of $n$ wires each—the $A_k$, the $B_k$ , and the $S_k$ —and also another wire $C$.

The number of `full-adder` we need to string together is the same as the length of any of the lists $A_k, B_k, S_k$. Thus we can recursively construct the circuit until we reach the last input the list.

```
(define (ripple-carry-adder Ak Bk Sk c-in)
  (let ((c-out (make-wire)))
    (full-adder (car Ak) (car Bk) c-in (car Sk) c-out)
    (if (not (null? (cdr Ak)))
        (ripple-carry-adder (cdr Ak) (cdr Bk) (cdr Sk) c-out))
    'ok))
```

Note that we have assumed that the initial signal of `(make-wire)` procedure is 0, for otherwise we have to run `(set-signal!  c-out 0)` if `(cdr Ak)` is null.

The time delay of our circuit is the sum of all `full-adder` in the circuit, which we denote as $nF_d$ where $F_d$ is the time delay of one `full-adder`. Each `full-adder` has a delay of $2nH_d{+}nI_d$, where $H_d, I_d$ are the time delays of `half-adder` and `inverter` respectively. In turn, the `half-adder` has a time delay of $2A_d{+}\max(O_d, A_d + I_d)$. Here we denote the `and-gate` by $A_d$ and $O_d$ is the `or-gate`. Hence the `ripple-carry-adder` will have a total time delay of

$$4nA_d + 2n \cdot \max(O_d, A_d + I_d) + nI_d.$$

#### Exercise 3.31

> The internal procedure `accept-action-procedure!` defined in `make-wire` specifies that when a new action procedure is added to a wire, the procedure is immediately run. Explain why this initialization is necessary. In particular, trace through the `half-adder` example in the paragraphs above and say how the system's response would differ if we had defined `accept-action-procedure!` as
>
> ```
> (define (accept-action-procedure! proc)
>   (set! action-procedures
>         (cons proc action-procedures)))
> ```

If we don't run the procedure immediately when creating the function box, then none of the wires' signal will change at this stage. This means that the signal in each wire may be wrong depending on the values it receives. Let's look at the `half-adder` definition

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

Assume that all wires $a, b, s$ and $c$ have signal 0. Then we expect that

1. `(or-gate a b d)` produces a 0 signal in wire $d$.
2. `(and-gate a b c)` produces a 0 signal in wire $c$.
3. `(inverter c e)` produces a 1 signal in wire $e$.
4. `(and-gate d e s)` produces a 0 signal in wire $s$.

All cases above are satisfied except 3). Since $e$ was created with `(make-wire)` it will be initialized to 0. But because we don't run the action procedure immediately, `(inverter c e)` will not change the signal in $e$. Hence $e$ will remain initialized to 0 when it should have been changed to 1.

This has consequences for the response we get from the `half-adder`. We know before that the output $s$ will be 1 whenever precisely one of $a$ or $b$ signal is 1. But this is no longer true. Because, if we now

```
(set-signal! a 1)
(propagate)
```

then $d$ is 1 due to `(or-gate a b d)`, $c$ will not change due to `(and-gate a b c)` and therefore `(inverter c e)` will not run its action procedure. So $e$ will remain

0 and therefore $s$ will be 0 because `(and-gate d e s)` will not change the output signal.

**Exercise 3.32**

> The procedures to be run during each time segment of the agenda are kept in a queue. Thus, the procedures for each segment are called in the order in which they were added to the agenda (first in, first out). Explain why this order must be used. In particular, trace the behavior of an `and-gate` whose inputs change from 0, 1 to 1, 0 in the same segment and say how the behavior would differ if we stored a segment's procedures in an ordinary list, adding and removing procedures only at the front (last in, first out).

Denote the input wires to the `and-gate` with $a, b$ and the output as $c$. Assume the `and-gate` is in the initial state `(a=0, b=1, c=0)` with an empty agenda. Whenever a signal is changed the `and-gate` will run the following procedure to determine the value of the output $c$.

```
(logical-and (get-signal a) (get-signal b))
```

After running the following instructions

```
(set-signal! a 1)
(set-signal! b 0)
(propagate)
```

the agenda is a list of containing these lines

```
(set-signal! c (logical-and 1 1))
(set-signal! c (logical-and 1 0))
```

that are executed after a delay. The state of the `and-gate` will be determined by the last action executed in the agenda.

If we process the agenda first-in, first out the last action in the agenda is `(logical-and 1 0)`. Hence, the final state is `(a=1, b=0, c=0)` which is what we expect.

On the other hand, if we process the agenda last-in, first-out the then the last action executed will be `(logical-and 1 1)` which sets the final state incorrectly to `(a=1, b=1, c=1)`.

### Exercise 3.33

> Using primitive `multiplier`, `adder`, and `constant` constraints, define a procedure `averager` that takes three connectors `a`, `b`, and `c` as inputs and establishes the constraint that the value of `c` is the average of the values of `a` and `b`.

We will need to create two additional internal connectors to wire the primitives together to build averager.

```scheme
(define (averager a b c)
  (let ((sum (make-connector))
        (const (make-connector)))
    (adder a b sum)
    (constant 0.5 const)
    (multiplier sum const c)))
```

**Exercise 3.34**

> Louis Reasoner wants to build a `squarer`, a constraint device with two terminals such that the value of connector `b` on the second terminal will always be the square of the value `a` on the first terminal. He proposes the following simple device made from a `multiplier`:
>
> ```
> (define (squarer a b)
>   (multiplier a a b))
> ```
>
> There is a serious flaw in this idea. Explain.

Suppose connector `a` has no value and we set connector `b` to 25. Since the relation is a square, we expect the system to give us the value of 5 for connector `a`. However, this does not happen. To understand this let's take a look at the `cond` of `mulitiplier`.

```
(cond ((or (and (has-value? m1) (= (get-value m1) 0))
           (and (has-value? m2) (= (get-value m2) 0)))
       ...)
      ((and (has-value? m1) (has-value? m2)) ...)
      ((and (has-value? product) (has-value? m1)) ...
      ((and (has-value? product) (has-value? m2)) ...)
```

Here `product` has a value but none of `m1` nor `m2` have. So no case in `cond` applies and `process-new-value` simply does nothing.

### Exercise 3.35

> Ben Bitdiddle tells Louis that one way to avoid the trouble in Exercise 3.34 is to define a squarer as a new primitive constraint. Fill in the missing portions in Ben's outline for a procedure to implement such a constraint:
>
> ```
> (define (squarer a b)
>   (define (process-new-value)
>     (if (has-value? b)
>         (if (< (get-value b) 0)
>             (error "square less than 0: SQUARER"
>                    (get-value b))
>             ⟨alternative1⟩)
>         ⟨alternative2⟩))
>   (define (process-forget-value) ⟨body1⟩)
>   (define (me request) ⟨body2⟩)
>   ⟨rest of definition⟩
>   me)
> ```

In case we have a value for connector `b`, then we need to set the value of connector `a` to the square root of `b`. On the other hand, if there's a value for `a` then we set the value of `b` to the square of `a`. The rest of the procedure is very similar to the other primitives we have already seen like `adder`.

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "square less than 0: SQUARER"
                   (get-value b))
            (set-value! a (sqrt (get-value b)) me))
        (set-value! b (square (get-value a)) me)))
  (define (process-forget-value)
    (forget-value! a me)
    (forget-value! b me))
  (define (me request)
    (cond ((eq? request 'I-have-a-value) (process-new-value))
          ((eq? request 'I-lost-my-value) (process-forget-value))
          (else (error "Unknown request: MULTIPLIER"
                       request))))
  (connect a me)
  (connect b me)
  me)
```

### Exercise 3.37

> Exercise 3.37: The `celsius-fahrenheit-converter` procedure is cumbersome when compared with a more expression-oriented style of definition, such as
>
> ```
> (define (celsius-fahrenheit-converter x)
>   (c+ (c* (c/ (cv 9) (cv 5))
>           x)
>       (cv 32)))
> (define C (make-connector))
> (define F (celsius-fahrenheit-converter C))
> ```
>
> Here `c+`, `c*`, etc. are the "constraint" versions of the arithmetic operations. For example, `c+` takes two connectors as arguments and returns a connector that is related to these by an adder constraint:
>
> ```
> (define (c+ x y)
>   (let ((z (make-connector)))
>     (adder x y z)
>     z))
> ```
>
> Define analogous procedures `c-`, `c*`, `c/`, and `cv` (constant value) that enable us to define compound constraints as in the converter example above.

We can define `cv` in terms of `constant` which is already supplied to us.

```
(define (cv value)
  (let ((x (make-connector)))
    (constant value x)
    x))
```

If we are a little clever we can define `c-` in terms of `adder`. If $x - y = z$ then we have that $x = y + z$, therefore

```
(define (c- x y)
  (let ((z (make-connector)))
    (adder y z x)
    z))
```

Multiplication is straight-forward with the use of `multiplier`.

```
(define (c* x y)
  (let ((z (make-connector)))
    (multiplier x y z)
    z))
```

By noticing that if $x/y = z$ then $x = zy$ whenever $y \neq 0$, we can define `c\` in terms of `multiplier`

```scheme
(define (c/ x y)
  (let ((z (make-connector)))
    (multiplier y z x)
    z))
```

Do note however, that setting the denominator `y` to 0 is undefined. Doing so in the current implementation will not give us a nice error message.

```scheme
(define x (make-connector))
(define y (make-connector))
(define z (c/ x y))
(probe "ratio" z)
(set-value! x 1 'user)
(set-value! y 0 'user)                  ; ERROR: Contradiction (1 0)
```

To get better error messages we need to define division as a primitive similar to how `multiplier` is defined in the book. In one of the `conds` we check if `b` is `zero` and error if so. We don't show this implementation here since it is very similar to `multiplier` in the book.