Environment Model
MIT 6.001 Recitation

Warm up

```
(define f
  (lambda(x)
    (lambda(y)
      (set! x (+ x y))
      x)))
(define g (f 4))
(g 3)
```

Factorial - iterative

Evaluate the expressions below, following the rules of the environment model. Draw the corresponding environment diagram.

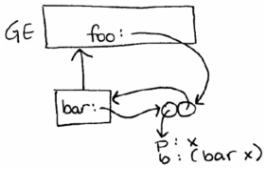
```
(define (fact n)
  (define (helper n result)
    (if (= n 1) result
      (helper (- n 1) (* n result))))
  (helper n 1)
(fact 3)
```

Let

```
(define x 10)
(let ((x 2)
      (y x))
  (* x y))
```

Challenge

what code made this environment diagram?




```
(define foo (let () (define bar (lambda(x)(bar x)))))
```

Fibonacci with counter

Recall the function fib-1 that takes an integer n and computes the n'th Fibonacci number.

```
(define fib-1 (lambda(n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib-1 (- n 1)) (fib-1 (- n 2)))))))
```

What is the Order of Growth in Time for the procedure fib-1? This is a tough one to figure out. Maybe this tree will help. Consider the number of recursive calls to fib-1 when the following is evaluated: (fib-1 5)

What if we want to see exactly how many times fib-1 is being called? Recall the function `make-count-proc` from last section? Write it again and add a 'reset function that restets the counter to zero.

```
(define make-count-proc
  (let ((counter 0))
    (lambda(x)
      (cond ((eq? x 'count) count)
            ((eq? x 'reset) (set! counter 0) counter)
            (else (set! counter (+ 1 counter)) (f x))))))
```

How can we use `make-count-proc` to define `fib-2` that will keep a count of the number of recursive calls? Be carefull, this is not so easy.

```
(define fib2 (make-count-proc fib1))
(define fib1 fib2) ;; if we don't do this, we don't recursively call the procedure with counter.
```

Take a look at these calls to fib-2:

`(fib-2 'reset)` ➔ `0` `(fib-2 30)` ➔ `832040` `(fib-2 'count)` ➔ `2692537`

That's pretty inefficient! We're recursively calling `fib-2` over and over again with the same argument and keep computing things we've already computed before. How can we fix this?

Fun question: how can you keep track of the number of procedures with counter that you have created?

Answer: (define make-count-proc (make-count-proc make-count-proc))

Memoization

Recall that the procedure make-count-proc takes in a procedure and returns a very similar procedure (from the caller's point of view), but this new procedure keeps some local state around and does something else each time it is called.

We want to write the procedure memoize that takes in a procedure of one argument and returns a procedure that keeps track of the all previously computed values. If a value passed in was passed in before, the procedure simply returns the saved value. What kind of data structure can you use to store the previous calls?

Association lists, hash tables, binary tress

Write the procedure memoize:

```
(define memoize (lambda f)
  (let ((history ()))
    (lambda(x)
      (let ((result (find-assoc x history)))
        (if result result
           (let ((y (f x)))
             (set! history (add-assoc x y history))
             y))))))
```

Now define `fib-3` that uses memoization and a counter (assuming we have not created fib2).

```
(define fib3 (memoize (make-count-proc fib1)))
(define fib1 fib2)
```

Challenge: Draw the Environment Diagram for the definition of `fib-3` (above) and then for the expression `(fib-3 3)`.

Reminder: association lists and hash tables

```
(define (find-assoc key alist)
   (cond
     ((null? alist) #f)
     ((equal? key (caar alist)) (cadar alist))
     (else (find-assoc key (cdr alist)))))

(define (add-assoc key val alist)
  (cons (list key val) alist))
```

Search for a maze

Remember the general search code? We want to use it to find the the solution of a maze. Below, we provide code for creating and accessing a maze. We then define a path abstraction that stores the list of pairs of coordinates x y in the maze. Finally, we need to define a state data structure to store both the path and the maze.

Study the code and discuss how to implement various search strategies and how they might behave. What is the strategy implemented below?

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; MAZE

(define (make-empty-maze width height)
  (let ((vec (make-vector (* width height) 'empty)))
    (list 'maze width height vec)))
(define (get-height maze) (caddr maze))
(define (get-width maze) (cadr maze))
(define (get-vector maze) (cadddr maze))
(define (get-index x y maze) (+ x (* y (get-width maze))))
   ;; x and y should be between 0 and width-1, 0 height-1
(define (empty-cell? x y maze)
  (eq? 'empty (vector-ref (get-vector maze) (get-index x y maze))))
(define (path-cell? x y maze)
  (eq? 'path (vector-ref (get-vector maze) (get-index x y maze))))
(define (set-full-cell! x y maze)
  (vector-set! (get-vector maze) (get-index x y maze) 'full))
(define (set-empty-cell! x y maze)
  (vector-set! (get-vector maze) (get-index x y maze) 'empty))
(define (set-path-cell! x y maze)
  (vector-set! (get-vector maze) (get-index x y maze) 'path))

(define (display-maze maze)
  (let ((width (get-width maze)) (height (get-height maze)))
    (define (helper x y)
      (cond ((>= y height) (newline))
            ((>= x width) (newline) (helper 0 (+ 1 y)))
            (else (cond ((empty-cell? x y maze) (display " "))
                        ((path-cell? x y maze) (display "X"))
                        (else (display "o")))
                  (helper (+ 1 x) y))))
    (helper 0 0)))
(define (make-boundary! maze)
 (let ((width (get-width maze)) (height (get-height maze)))
   (define (make-horizontal! x)
     (if (< x width) (begin (set-full-cell! x 0 maze)
                            (set-full-cell! x (- height 1) maze)
                            (make-horizontal! (+ x 1)))))
   (define (make-vertical! y)
     (if (< y height) (begin (set-full-cell! 0 y maze)
                             (set-full-cell! (- width 1) y maze)
                             (make-vertical! (+ y 1)))))
   (make-horizontal! 0)
   (make-vertical! 0)))

(define (make-random-walls! n maze)
 (let ((width (get-width maze)) (height (get-height maze)))
  (define (helper k)
    (if (< k n) (begin
                  (set-full-cell! (random width) (random height) maze)
                  (helper (+ 1 k)))))
   (helper 0)))
(define (make-compliant! maze)
 (let ((width (get-width maze)) (height (get-height maze)))
   (set-empty-cell! 1 1 maze)
   (set-empty-cell! (- width 2) (- height 2) maze)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; PATH

(define (make-empty-path) ())
(define (add-step x y path) (cons (cons x y) path))
(define (get-last-x path) (caar path))
(define (get-last-y path) (cdar path))
```

```
(define (rest path) (cdr path))
(define (empty-path? path) (null? path))

(define (embed-path-in-maze! path maze)
  (if (not (empty-path? path))
      (begin
        (set-path-cell! (get-last-x path) (get-last-y path) maze)
        (embed-path-in-maze! (rest path) maze))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; STATE
(define (make-state path maze) (cons path maze))
(define (get-path state) (car state))
(define (get-maze state) (cdr state))

(define (make-initial-state maze)
  (let ((path (make-empty-path)))
    (make-state
     (add-step (- (get-width maze) 2) (- (get-height maze) 2) path)
     maze)))

(define (next-moves state)
  (let* ((path (get-path state))
         (maze (get-maze state))
         (x (get-last-x path))
         (y (get-last-y path))
         (tentative (list (cons (+ 1 x) y)
                          (cons x (+ 1 y))
                          (cons (- x 1) y)
                          (cons x (- y 1)))))
    (map (lambda(p)(make-state (add-step (car p) (cdr p) path) maze))
         (filter (lambda(p)(empty-cell? (car p) (cdr p) maze)) tentative))))

(define (done-with-maze? state)
  (let* ((path (get-path state)))
    ;(display path) (newline)
    (and (= (get-last-x path) 1)
         (= (get-last-y path) 1))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  SEARCH

(define (search start-state done? succ-fn merge-fn)
  (define (search1 queue)
    (if (null? queue)
        #f
        (let ((current (car queue)))
          (if (done? current)
              current
              (search1
               (merge-fn (succ-fn current)
                         (cdr queue)))))))
  (search1 (list start-state)))

(define (solve maze)
  (let ((state
         (search (make-initial-state maze) done-with-maze? next-moves
                 (lambda(x y)(append y x)))))
    (if state (get-path state) (error "no maze solution found"))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; FUN

(define maze (make-empty-maze 8 8))
(make-boundary! maze)
(make-random-walls! 10 maze)
(make-compliant! maze)
(display-maze maze)
(define path (solve maze))
(embed-path-in-maze! path maze)
(display-maze maze)
```