MIT 6.001 recitation: review for quiz 2

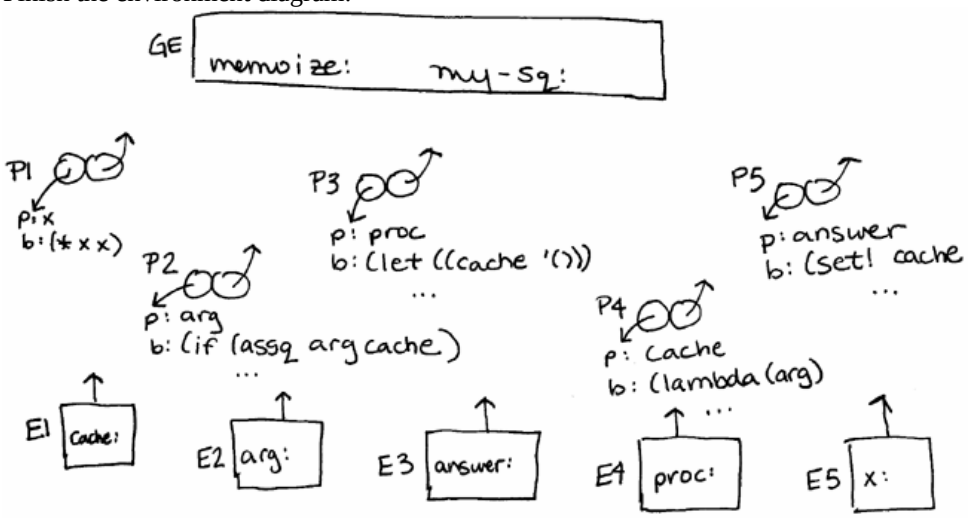
The quiz covers everything until the environment model included.

Memoization

Memoization is a clever idea that allows us to save on computation. It works by keeping track of evaluation of a procedure on a specific argument, and simply returns the remembered value if the procedure has already been run on that argument.

```
(define (memoize proc)
  (let ((cache '()))
    (lambda (arg)
      (if (assq arg cache)
          (cadr (assq arg cache))
          (let ((answer (proc arg)))
            (set! cache (cons (list arg answer) cache))
            answer)))))

(define my-sq (memoize (lambda (x) (* x x))))
(my-sq 5)
```

Finish the environment diagram:



To what does the environment pointer of P1 point?
To what does the environment pointer of P2 point?
To what does the environment pointer of P3 point?
To what does the environment pointer of P4 point?
To what does the environment pointer of P5 point?

What is the enclosing environment for E1?
What is the enclosing environment for E2?
What is the enclosing environment for E3?
What is the enclosing environment for E4?
What is the enclosing environment for E5?

To what is the variable memoize in the global environment bound?
To what is the variable my-sq in the global environment bound?
To what is the variable cache in E1 bound?
To what is the variable arg in E2 bound?

What variable is bound to the procedure object P1? Give both the name and the environment.
What variable is bound to the procedure object P5? Give both the name and the environment.

Insertion sort

* let's write a list that is maintained sorted thanks to insertion sort:

```
(define lst (make-sorted-list))
(get-data lst)  ->  ()
(insert lst 4)
(insert lst 1)
(get-data lst)  ->  (1 4)
(insert lst 5)
(insert lst 6)
(insert lst 7)
(insert lst 3)
(get-data lst)  ->  (1 3 4 5 6 7)

(define (make-sorted-list)
  (cons 'sorted '()))

(define (get-data sorted-list)
  (if (eq? (car sorted-list) 'sorted)
        (cdr sorted-list)
        (error "not a sorted-list")))

(define (insert sorted-list element)
       YOUR CODE HERE
```

* whats the order of growth of insertion?

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))
```

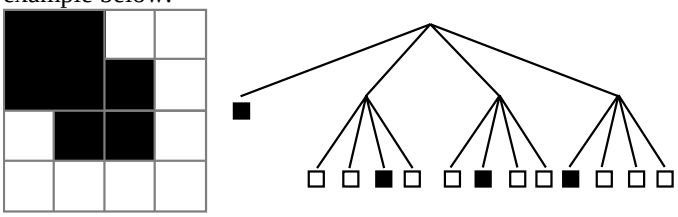
```
(define (get-last-y path) (cdar path))
(define (rest path) (cdr path))
(define (empty-path? path) (null? path))

(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)
```

Quadtrees

From Spring 2005 final exam

A Quadtree is a popular data structure in computer graphics that allows for the encoding of spatial information using recursive subdivisions of a square. In this problem, we define a quadtree abstract datatype to represent black-and-white bitmap, where a pixel is either white or black.

The leaves of a quadtree are either black or white, and other nodes have exactly four children representing the four quadrants of the subdivision of the square in the order North-West, North-East, South-West, and South-East. See the example below:



We provide the following abstraction:

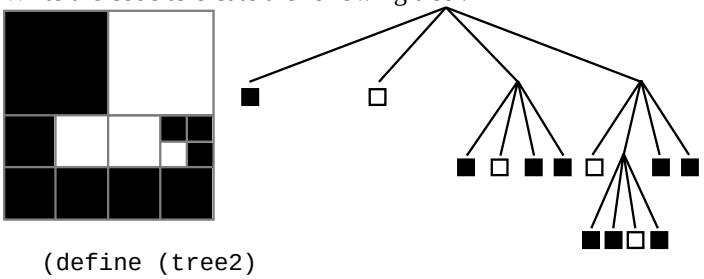
```
(define (black-qt) 'black)
(define (white-qt) 'white)
(define (node-qt nw ne sw se) (list nw ne sw se))

(define (qt-black? qt) (eq? qt 'black))
(define (qt-white? qt) (eq? qt 'white))
(define (qt-node? qt) (list? qt))

(define (qt-nw qt) (car qt))
(define (qt-ne qt) (cadr qt))
(define (qt-sw qt) (caddr qt))
(define (qt-se qt) (cadddr qt))
```

For example, the above tree can be obtained using

```
(define tree1
          (let ((ne (node-qt (white-qt) (white-qt) (black-qt) (white-qt)))
                                        (sw (node-qt (white-qt) (black-qt) (white-qt)
(white-qt)))
                                        (se (node-qt (black-qt) (white-qt) (white-qt)
(white-qt)))
                        (node-qt (black) ne sw se))
```

Question 1

Write the code to create the following tree :



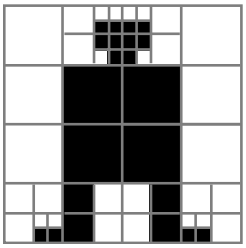
```
(define (tree2)
          YOUR-CODE-HERE
```

Question 2: Depth-first encoding

We want to represent quadtrees in a compact and flat manner using a list composed of '*n*' to indicate a node, *'0'* for black, and *'1'* for white. More precisely, we define the *depth-first encoding* of the quadtree as:

- 0 if the tree is a black leaf, and 1 if it is a white leaf
- the concatenation of 'n and the depth-order encoding of the sub-trees in the order described above (NW, NE, SW, SE) otherwise.

For example, the following tree



shoudl return
(n n 0 n 0 n 0 0 1 1 0 n 1 1 0 1 0 1 n n n 0 0 1 1 0 n 1 1 1 0 0 0 1 0 n 0 1 n 0 0 0 n 0 0 0 n 0 0 1 1 n 1 0 1 0 n 1 0 n 0 1 0 1 n 0 0 n 0 0 1 1 0)

For this, we will write a function qt-depth-first-encode, and your job will be to fill in the code for CODE-1, CODE-2, and CODE-3. Be careful to respect the abstraction barrier.

```
(define (qt-depth-first-encode qt)
        (cond  (  (qt-black? qt) CODE-1
                  CODE-2
                  (else CODE-3))))
```

Question 3: Simplification

Some quadtrees contain redundant information when subdivision was used in regions of constant color. We want to simplify such tree into the most-compact quadtree. We will do this in two steps: first, we simplify nodes that have only leaves as children, next we will recursively simplify the full tree.
First, we want to simplify a node that has four black children or four white children. Write a function qt-conflate that takes a node and verifies if its four children are all black (resp all white), in which case it returns a black (resp. white) leaf. Otherwise, return the node unchanged.

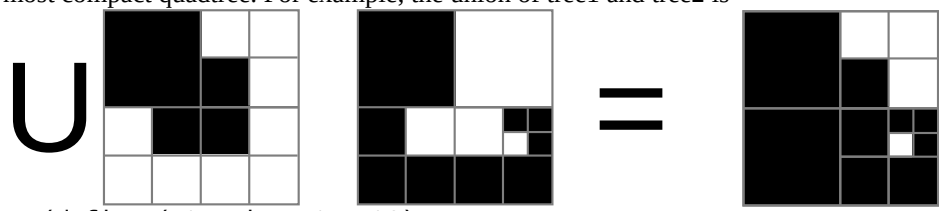
```
(define (qt-conflate n)
          YOUR-CODE
```

Next, we want to recursively simplify the tree using the qt-conflate procedure above.

```
(define (qt-simplify qt)
          (if qt-node? qt)
                    YOUR-CODE-HERE
          qt))
```

Question 4: Union

We want to take two images described by a quadtree and compute the union of the black parts. Note that the subdivision structure of the union quadtree might be simpler than that of the two sub-trees. In this case, we want the most compact quadtree. For example, the union of tree1 and tree2 is




```
(define (qt-union qt1 qt2)
          (cond
                ((qt-black? qt1) YOUR-CODE-HERE
                ((qt-white? qt1) YOUR-CODE-HERE
                ((qt-black? qt2) YOUR-CODE-HERE
                ((qt-white? qt2) YOUR-CODE-HERE
                (else
                      YOUR-CODE-HERE)
```