Question 1: Lists are fun!

Someone wrote this procedure list-fun.

(define (list-fun x)
  (if (or (null? x) (not (list? x)))
      (list x)
      (let ((elem (car x)))
        (map (lambda (y) (cons elem (list-fun y))) x))))

a) What does the procedure list-fun return for x = a

(list-fun 'a) ➔ (a)

b) What does the procedure list-fun return for x = '(a)

(list-fun '(a)) ➔ ((a a))

c) What does the procedure list-fun return for x = '(a b c)

(list-fun '(a b c)) ➔ ((a a) (a b) (a c))

d) What is the length of the list returned for x = '(((a) (b) (c)))

(length (list-fun '(((a) (b) (c)))) ➔ 3
(list-fun '(((a) (b) (c))) ➔ (((a) (a a)) ((a) (b b)) ((a) (c c)))

e) What is the domain of list-fun?

Anything, lists are treated specially, but everything else is packaged into a list
Question 2: Party!
To plan an upcoming party you make an association list of ingredients that you need and their cost.

(define simple-grocery-L '((cake-mix 2) (eggs 3) (soda 2)))

But what would be really cool is if you could figure out how much a cake, which is made up of cake-mix and eggs, would cost. You’ve made up a more complicated version of your old grocery list. It has both individual items and their costs AND composite items and their ingredients. For example:

(define complex-grocery-L
  '(
    (cake cake-mix eggs)
    (strawberry-shortcake cake strawberries whipped-cream)
    (cake-mix 2)
    (eggs 3)
    (strawberries 4)
    (whipped-cream 3)
    (soda 5)
    (salsa 3)
    (chips 4))))

Now you can write a procedure to calculate the total cost of your menu. For example:

(total-cost '(chips soda) complex-grocery-L) \(\rightarrow\) 9
(total-cost '(cake) complex-grocery-L) \(\rightarrow\) 5
(total-cost '(strawberry-shortcake) complex-grocery-L) \(\rightarrow\) 12
(total-cost '(strawberry-shortcake chips soda) complex-grocery-L)
\(\rightarrow\) 21

(define (total-cost menu grocery-info)
  (if (null? menu)
      0
      (let ((ingred (cdr (assoc (car menu) grocery-info))))
        (if (number? (car ingred))
            (+ (car ingred)
                (total-cost (cdr menu) grocery-info)))
            (+ (total-cost ingred grocery-info)
                (total-cost (cdr menu) grocery-info))))))

NOTE: short version (above), long version that we went over during the review session (below)
(define (total-cost menu grocery-info)
  (reduce + (map (lambda (food) (cost-of-item food grocery-info))
                menu)))

(define (cost-of-item food grocery-info)
  (let ((ingred (cdr (assoc food grocery-info))))
    (if (number? (car ingred))
        (car ingred)
        (total-cost ingred grocery-info))))

;; note the last line is different from how I did it during
;; the review session, but I was essentially copying the
;; total-cost procedure.

Question 3: Fractals
We’ve got a cool new fractal for you to try! It has ovals in it – so here is a helper
procedure to draw ovals. The new fractal is called mystery-fractal. The squares
are NOT part of the fractal! They are just to show scale! The scale is the same in
each image.

(define (draw-white-oval x1 y1 x2 y2)
  (draw-oval x1 y1 x2 y2 'fill 'white))
a) Draw the picture for mystery-fractal for $n=2$.

b) Fill in the blank for $n$ in the 4th picture

```
(define (mystery-fractal x1 y1 x2 y2 n)
  (if (< n 0)
      'done
      (let ((xmid (/ (+ x1 x2) 2))
            (ymid (/ (+ y1 y2) 2))
            (mystery-fractal
              0 0 200 200 0)
            (mystery-fractal
              0 0 200 200 1))
      (mystery-fractal
        0 0 200 200 2)
      (mystery-fractal
        0 0 200 200 __))
  )
)
```
(xplus (+ x2 (/ (- x2 x1) 2)))
(yplus (+ y2 (/ (- y2 y1) 2))))
(draw-white-oval x1 y1 x2 y2)
(mystery-fractal x1 y1 xmid ymid (- n 1))
(mystery-fractal x2 y1 xplus ymid (- n 1))
(mystery-fractal x1 y2 xmid yplus (- n 1))

))))