Sample midterm 1 #2

Problem 1 (What will Scheme print?).

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just say "error"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just say "procedure"; you don't have to show the form in which Scheme prints procedures.

```
(every (bf x) '(ab cd ef gh))
```

```
(cond ('hello 5) (#t 6) (else 7))
```

```
(let ((x 10)
      (y (+ x 2)) )
  (* y 3))
```

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. **Also, draw a box and pointer diagram for the value produced by each expression.**

```
(cons (list '() '(b)) (append '(c) '(d)))
```

```
((lambda (x) (cons x x)) '(a))
```

```
(cdar '((1 2) (3 4)))
```

Problem 2 (Orders of growth, iterative/recursive processes).

```
(define (garply n)
  (if (< n 20)
      n
      (+ (foo n)
         (garply (- n 1)) )) )
```

Assuming `foo` is defined somewhere, please **circle** True or False, and **in one sentence** explain your choice.

True or False: We have enough information to determine the order of growth of `garply`.

True or False: No matter how `foo` is defined, `garply` will always have an order of growth greater than or equal to $\Theta(n)$.

True or False: `garply` has an order of growth in $\Theta(n^2)$ if `foo` is defined as:

```
(define (foo n)
  (if (< n 100)
      121
      (+ (* n 100) (foo (- n 1)))))
```

True or False: `garply` generates a iterative process.

Problem 3 (Normal/applicative order).

If an expression produces an error, just say "error"; if it returns a procedure, just say "procedure."

Given the following definitions:

```
(define (mountain x) 'done)
(define (dew) (dew))
```

(a) What will be the result of the expression `(mountain (dew))`

in normal order? ______________________________

in applicative order? ______________________________

(b) What will be the result of the expression `(mountain dew)`

in normal order? ______________________________

in applicative order? ______________________________

Problem 4 (Recursive procedures).

Write a procedure `every-nth` that takes two arguments, a number n and a sentence. It should return the sentence formed by choosing every nth element of the sentence. For example:

```
> (every-nth 3 '(the rain in spain stays mainly on the plain))
(in mainly plain)

> (every-nth 2 '(in the town where i was born lived a man who sailed to sea))
(the where was lived man sailed sea)

> (every-nth 4 '(you think you lost your love well i saw her yesterday))
(lost i)
```

Your procedure should work for sentences of any length.

Problem 5 (Higher order procedures).

Here are two procedure definitions with examples of their use:

```
(define (differences sent)
  (if (empty? (bf sent))
      '()
      (se (- (first sent) (first (bf sent)))
          (differences (bf sent)))))

> (differences '(86 42 15 9))
(44 27 6)
> (differences '(10 20 5))
(-10 15)

(define (wordpairs sent)
  (if (empty? (bf sent))
      '()
      (se (word (first sent) (first (bf sent)))
          (wordpairs (bf sent)))))

> (wordpairs '(now here after math))
(nowhere hereafter aftermath)
> (wordpairs '(fat her mit e rupt ure))
(father hermit mite erupt rupture)
```

Write a procedure `pairmap` that generalizes the pattern followed by these two examples.
⇒ **Then rewrite `differences` and `wordpairs` using your `pairmap`.**

Problem 6 (Data abstraction).

This two-part question is about an abstract data type called "sockdrawer," representing a dresser drawer full of socks.

(a) Suppose we represent the socks in the drawer as a list of names of colors, like `(blue blue blue brown grey grey brown blue)`. You are given the selectors `colors` and `howmany`:

```
(define (colors sockdrawer)
  (define (remdup seq)
    (cond ((null? seq) '())
          ((memq (car seq) (cdr seq)) (remdup (cdr seq)))
          (else (cons (car seq) (remdup (cdr seq)))) ))
  (remdup sockdrawer) )

(define (howmany color sockdrawer)
  (length (filter (lambda (sock) (eq? sock color)) sockdrawer)) )

> (colors '(blue blue blue brown grey grey brown blue))
(grey brown blue)

> (howmany 'blue '(blue blue blue brown grey grey brown blue))
4
```

Write the predicate `odd-sock?` that takes a sockdrawer as its argument, and returns `#t` if any color in the drawer has an odd number of socks. Respect the data abstraction.

(b) Now suppose we decide to change the internal representation of a sockdrawer from an unordered list of color names to a list of lists in this format:

```
((blue 4) (brown 2) (grey 2))
```

Rewrite the selectors `colors` and `howmany` to reflect this new internal representation.