Sample midterm 1 #1

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 - (keep number? '(the 1 after 909)))
```

```
((lambda (a b) ((if (< b a) + *) b a)) 4 6)
```

```
(word (first '(cat)) (butlast 'dog))
```

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 1 2) (cons 3 4))
```

```
(let ((p (list 4 5)))
  (cons (cdr p) (cddr p)) )
```

```
(cadadr '((a (b) c) (d (e) f) (g (h) i))
```

Problem 2 (Orders of growth).

(a) Indicate the order of growth in time of `foo` below:

```
(define (foo n)
  (if (< n 2)
      1
      (+ (baz (- n 1))
         (baz (- n 2)) ) ))

(define (baz n)
  (+ n (- n 1)) )
```

___$\Theta(1)$ ___$\Theta(n)$ ___$\Theta(n^2)$ ___$\Theta(2^n)$

(b) Indicate the order of growth in time of `garply` below:

```
(define (garply n)
  (if (= n 0)
      0
      (+ (factorial n) (garply (- n 1)))))

(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

___$\Theta(1)$ ___$\Theta(n)$ ___$\Theta(n^2)$ ___$\Theta(2^n)$

Problem 3 (Normal and applicative order).

Imagine that there is a **primitive** procedure called `counter`, with no arguments, that returns 1 the first time you call it, 2 the second time, and so on. (The multiplication procedure `*`, used below, is also primitive.)

Supposing that `counter` hasn't been called until now, what is the value of the expression

```
(* (counter) (counter))
```

under applicative order? _____

under normal order? _____

Problem 4 (Iterative and recursive processes).

One or more of the following procedures generates an iterative process. Circle them. Don't circle the ones that generate a recursive process.

```
(define (butfirst-n num stuff)
  (if (= num 0)
      stuff
      (butfirst-n (- num 1) (bf stuff))))
```

```
(define (member? thing stuff)
  (cond ((empty? stuff) #f)
        ((equal? thing (first stuff)) #t)
        (else (member? thing (bf stuff)))))
```

```
(define (addup nums)
  (if (empty? nums)
      0
      (+ (first nums)
         (addup (bf nums)))))
```

Problem 5 (Recursive procedures).

Write a function `SYLLABLES` that takes a word as argument and returns the number of syllables in the word. For our purposes, the number of syllables is equal to the number of vowels, except that consecutive vowels only count as one syllable:

```
> (syllables 'banana)
3

> (syllables 'aardvark)
2

> (syllables 'cloud)
1
```

(In real life there are additional complications, like `Y` being a vowel sometimes, and silent `E`s not adding a syllable, but ignore these problems.) Hint: You may find it useful to write a function that chops off consecutive vowels from the beginning of a word.

You may assume the following definition of `vowel?`:

```
(define (vowel? letter)
  (member? letter '(a e i o u))
```

Problem 6 (Higher order functions).

For homework you wrote a procedure `ordered?` that takes a sentence of numbers as its argument, returning `#t` if the numbers are in ascending order. You used the primitive predicate `<` to compare the first number with the second.

It's possible to ask about the ordering of sentences of things other than numbers, provided we have some way of asking whether one thing comes before another. For example, we can ask whether the words in a sentence are in alphabetical order, or whether the *lengths* of the words are in increasing order.

This problem continues on the next page.

(Part a) Write a procedure `in-order?` that takes two arguments: first, a predicate function of two arguments that returns `#t` if its first argument comes before its second; second, a sentence. Your procedure should return `#t` if the sentence is in increasing order according to the given predicate. Examples:

```
> (define (shorter? a b)
    (< (count a) (count b)) )

> (in-order? shorter? '(i saw them standing together))
#t

> (in-order? shorter? '(i saw her standing there))
#f

> (in-order? < '(2 3 5 5 8 13))
#f

> (in-order? <= '(2 3 5 5 8 13))
#t

> (in-order? > '(23 14 7 5 2))
#t
```

(Part b) Write a procedure `order-checker` that takes as its only argument a predicate function of two arguments. Your procedure should return *a predicate function* with one argument, a sentence; this returned procedure should return `#t` if the sentence is in ascending order according to the predicate argument. For example:

```
> (define length-ordered? (order-checker shorter?))

> (length-ordered? '(i saw them standing together))
#t

> (length-ordered? '(i saw her standing there))
#f

> ((order-checker <) '(2 3 5 5 8 13))
#f

> ((order-checker <=) '(2 3 5 5 8 13))
#t

> ((order-checker >) '(23 14 7 5 2))
#t
```

Problem 7 (Data abstraction).

We want to write a program that uses the time of day as an abstract data type. We'll represent times internally as a list of three elements, such as `(11 23 am)` for 11:23 am. For the purposes of this problem, assume that the hour part is never 12, so there's never any special problems about noon and midnight. The hour will be a number 1–11, the minute will be a number 0–59, and the third element (which we'll call the *category*) must be the word `am` or the word `pm`. Here's our implementation:

```
(define (make-time hr mn cat) (list hr mn cat))
(define hour car)
(define minute cadr)
(define category caddr)
```

(a) This is a good internal representation, but not a good representation for the user of our program to see. Write a function `time-print-form` that takes a time as its argument and returns a word of the form `3:07pm`.

(b) If we want to ask whether one time is before or after another, it's convenient to use the 24-hour representation in which 3:47 pm has the form `1547`. Write a procedure `24-hour` that takes a time as its argument and returns the number that represents that time in 24-hour notation:

```
> (24-hour (make-time 3 47 'pm))
1547
```

Respect the data abstraction!

(c) Now we decide to change the *internal* representation of times to be a number in 24-hour form. But we want the constructor and selectors to have the same interface so that programs using the abstract data type don't have to change. Rewrite the constructor and selectors to accomplish this.