Sample midterm 1 #3

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.

```
(first '(help!))
```

```
((word 'but 'first) 'plop)
```

```
(let ((+ -)
      (- +))
  (- 8 2))
```

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.**

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

```
(filter (lambda (x) (if (list? x) (pair? x) (number? x)))
        '(1 () (2 3) (so) what))
```

```
(cddadr '((a b c d e) (f g h i j) (l m n o p) (q r s t u)))
```

Problem 2 (Orders of growth).

Suppose that you have a procedure `(f n)` that requires time $\Theta(n)$, and a procedure `(g n)` that requires time $\Theta(n^2)$. What is the order of growth required for a program to compute `(* (f n) (g n))`? **There may be more than one correct answer; check all that are correct.**

_____A. $\Theta(n)$ _____B. $\Theta(n^2)$

_____C. $\Theta(n + n^2)$ _____D. $\Theta(n^3)$

Problem 3 (Iterative/recursive processes).

Consider the function defined as follows:

$$\begin{aligned} \lg(1) &= 0 \\ \lg(n) &= \lg(\lfloor n/2 \rfloor) + 1, \qquad n > 1 \end{aligned}$$

In this problem you will write two Scheme procedures to compute this function. **Use the algorithm shown above;** do not use mathematical functions not shown here, such as `expt` or `sqrt`. You may use `(floor x)` to compute $\lfloor x \rfloor$.

(a) Write a procedure that computes this function using a recursive process.

(b) Write a procedure that computes this function using an iterative process.

Problem 4 (Normal/applicative order).

Given the primitive procedure `random` discussed in lecture, and the procedure `square` defined as follows:

```
(define (square x)
  (* x x))
```

Which of the following expresions may have a different value depending on whether applicative order or normal order is used? **There may be more than one correct answer; check all that are correct.**

A. `(random 10)`

B. `(* (random 10) (random 10))`

C. `(square (random 10))`

D. `(random (square 10))`

Problem 5 (Recursive procedures).

Mad Libs^{TM} is a campfire game in which players fill in blank places in a story, thereby creating a hilarious new story. We wish to implement this in Scheme.

`Mad-libs` is a procedure that takes a *sentence* `story` as its argument. Some of the words in `story` begin with '`*`', meaning they are to be replaced. `Mad-libs` returns a procedure that takes *two sentences,* `nouns` and `adjectives`, and replaces, in order, words from `story` beginning with '`*`' with words from `nouns` if the word is `*NOUN` or from `adjectives` if the word is `*ADJECTIVE`. For example, an infamous author once wrote:

It was a dark and stormy night... and the rain fell in torrents – except at occasional intervals, when it was checked by a violent gust of wind which swept up the streets (for it is in London that our scene lies), rattling along the housetops, and fiercely agitating the scanty flame of the lamps that struggled against the darkness. —Paul Clifford (1830)

```
(define my-story
   (mad-libs '(It was a *ADJECTIVE and *ADJECTIVE *NOUN)))

> (my-story '(night) '(dark stormy))
(It was a dark and stormy night)

> (my-story '(midterm) '(fun easy))
(It was a fun and easy midterm)
```

Write `mad-libs`. You may assume that the lengths of `nouns` and of `adjectives` are exactly the same as the number of words of that category in `story`.

Problem 6 (Higher order procedures).

We are going to generalize the idea of

```
(define (plural wd)
  (word wd 's))
```

by allowing any prefix and/or suffix to be added to a word. Here are some examples:

```
> (define plural (word-maker '(* s)))
> (plural 'book)
books

> (define past (word-maker '(* ed)))
> (past 'walk)
walked

> ((word-maker '(re *)) 'write)
rewrite

> ((word-maker '(de * ing)) 'construct)
deconstructing

> ((word-maker '(* -vs- *)) 'spy)
spy-vs-spy
```

`Word-maker` takes a sentence, called a *template*, as its argument. It returns a function that takes a word as argument and returns another word based on the template, but with any `*` in the template replaced with the argument word, and the result scrunched into a word. Write `word-maker`.

Don't use recursion; use higher-order functions. Here's a helper function for you:

```
(define (scrunch sent)
  (if (empty? sent)
      ""
      (word (first sent) (scrunch (butfirst sent)))))

> (scrunch '(here there and everywhere))
herethereandeverywhere
```

Problem 7 (Data abstraction).

An *association list* is a list of pairs, each of which associates a name (the *key*) with a value. For example, the association list

```
((a . 3) (b . 7) (c . foo))
```

associates the key `A` with the value `3`, the key `B` with the value `7`, and the key `C` with the value `FOO`.

Here are a constructor and two selectors for an *association* abstract data type:

```
(define make-association cons)
(define association-key car)
(define association-value cdr)
```

Note that this is an ADT for an association, not for an association list, which is just a sequence of associations.

(a) Scheme provides the procedure `assoc` for looking up a key in an association list. It returns the entire association, not just the value. Here is an implementation of `assoc`:

```
(define (assoc key a-list)
  (cond ((null? a-list) #f)
        ((equal? key (caar a-list)) (car a-list))
        (else (assoc key (cdr a-list)))))
```

(Both `caar` and `car` in the next-to-last line above are correct!)

Rewrite `assoc` to use the association ADT defined above. Don't make any unnecessary changes; your program should look much like the one above, but with some procedure calls renamed.

(Problem continues on the next page.)

(b) We are given an association list in which the keys are names of musical groups and the values are *sentences* of names of the group members, like this:

```
((Beatles . (John Paul George Ringo))
 ((Buffalo Springfield) . (Steve Neil Ritchie Dewey Bruce))
 (Who . (Pete Alec Roger Keith))
```

What follows is a procedure to make a "backwards" association list in which the keys are the musicians and the values are their groups, like this example:

```
((John . Beatles) (Paul . Beatles) (George . Beatles) (Ringo . Beatles)
 (Steve . (Buffalo Springfield)) (Neil . (Buffalo Springfield))
 (Richie . (Buffalo Springfield)) (Dewey . (Buffalo Springfield))
 (Bruce . (Buffalo Springfield)) (Pete . Who) (Alec . Who)
 (Roger . Who) (Keith . Who))
```

(Note to '60s rock fans: I've listed John Alec Entwistle by his middle name so that there won't be two people named John in this example. You can ignore the possibility of people with the same name; assume that won't happen.)

Here is the program:

```
(define (index groups)
  (if (null? groups)
      '()
      (append (index-one (car groups)) (index (cdr groups)))))

(define (index-one group)
  (define (help groupname people)
    (if (null? people)
        '()
        (cons (cons (car people) groupname)
              (help groupname (cdr people)))))
  (help (car group) (cdr group)))
```

Rewrite `index` and `index-one` to respect both the association ADT and the sentence ADT. Don't make any unnecessary changes.