Sample Exam 2

In this sample exam are some problems of types you might see on the second exam. Note that the exam will be a bit longer, with 7-8 problems.

Problem 1:

For each of the following sets of expressions, write down what the last one will return. If it returns an error, or doesn't return at all, say so. You can assume that each set of expressions is evaluated in a fresh Scheme buffer. Show your work to be eligible for partial credit.

Set 1:  \( \text{(define } f \ (\lambda \ (y) \ (\lambda \ (x) \ (\text{set}! \ x \ (+ \ x \ y)) \ x)) \) \)

\( \text{(define } g \ (f \ 3)) \)

\( g \ 4 \)

\( g \ 5 \)

Set 2:  \( \text{(define } f \ (\lambda \ (x) \ (\lambda \ (y) \ (\text{set}! \ x \ (+ \ x \ y)) \ x)) \) \)

\( \text{(define } g \ (f \ 3)) \)

\( g \ 4 \)

\( g \ 5 \)
Problem 2:

Assume the following Scheme expressions are evaluated sequentially. Fill in the table below, using eq?, eqv? and equal? to compare the items in the first two columns. You may find it useful to draw the box and pointer diagrams for this problem, but you are not required to do so.

(define a (list 1 2 3))
(define b (list 1 2 3))
(define c (list 4))
(define d (cons 4 a))
(define e (append c b))

<table>
<thead>
<tr>
<th></th>
<th>eq?</th>
<th>eqv?</th>
<th>equal?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cdr d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cdr e)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cdr d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3:

Write a procedure called `double-list` that multiples each element of a list by 2 in the two different ways below. For both versions, the same answer should be returned. For example, `(double-list '(1 2 3))` will return `(2 4 6)`.

a. Use the `scale-list` procedure below in your definition of `double-list`.

```scheme
(define (scale-list items factor)
  (if (null? Items)
      nil
    (cons (* (car items) factor)
          (scale-list (cdr items) factor)))
)
```

b. Use the `map` procedure below in your definition of `double-list`.

```scheme
(define (map proc items)
  (if (null? Items)
      nil
    (cons (proc (car items))
          (map proc (cdr items))))
)
```
Problem 4:

Write count-times, a procedure of two arguments, s and tree, that returns the number of times the first argument (an atom) appears in the second (a tree). Use the accumulate-tree procedure, given below, to write your solution.

(define (accumulate-tree tree term combiner null-value)
  (cond ((null? tree) null-value)
        ((not (pair? tree)) (term tree))
        (else (combiner (accumulate-tree (car tree) term combiner null-value)
                        (accumulate-tree (cdr tree) term combiner null-value))))

For example, (count-times 3 '((1 2 (3)) (4 3 ((3)) (5 3 6)))) would return 4 because 3 appears four times in the tree.
Problem 5:

Below is a definition for make-inc, a function that creates an accumulator that uses message passing.

```
(define (make-inc init)
  (let ((value init))
    (define (inc-val x)
      (set! value (+ value x))
      value)
    (define (dispatch m)
      (cond ((eq? m 'inc-val) inc-val)
            (else (error "Invalid message – MAKE-INC" m))))
    dispatch))
```

a) Change the code to add a new functionality. We’d like to be able to call the accumulator with the message ‘reset-val to reset the counter to 0.

You do not need to rewrite all of the code above. If you want to insert code into the function above, write it here and clearly indicate where it should be put in the code above. (Space was left in the function definition to allow you to do this.)
Problem 5 (continued):

Code is repeated here for your convenience.

(define (make-inc init)
  (let ((value init))
    (define (inc-val x)
      (set! value (+ value x))
      value)
    (define (dispatch m)
      (cond ((eq? m 'inc-val) inc-val)
            (else (error "Invalid message – MAKE-INC" m)))
      dispatch))

b) Now change the code to add another new functionality. We’d like to be able to call
the accumulator with the message ’set-val to reset the counter to a value passed in by
the user.

You do not need to rewrite all of the code above. If you want to insert code into the
function above, write it here and clearly indicate where it should be put in the code
above. (Space was left in the function definition to allow you to do this.)