

Sample Exam 2

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: (define f (lambda (y) (lambda (x) (set! x (+ x y)) x)))
(define g (f 3))
(g 4)
(g 5)

Set 2: (define f (lambda (x) (lambda (y) (set! x (+ x y)) x)))
(define g (f 3))
(g 4)
(g 5)

Set 3: (define x '(a b c))
(define y '(d e f))
(define z (append x y))
(set-car! x 'foo)
(set-car! y 'bar)
z

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))
```

		eq?	eqv?	equal?
a	b			
d	e			
(cdr d)	(cdr e)			
(cdr d)	a			

Problem 3:

Draw the box-and-pointer diagrams for r, s, x, and y that will result from evaluating the following expressions:

```
(define r (list 'a 'b))  
(define s (list 'c 'd))  
(define x (append r s))  
(define y (append r s))  
(set-car! (cdr x) 'e)  
(set-cdr! (cdr (cdr y)) (list r))
```

Problem 4:

Using the following procedure for dealing with trees,

```
(define (tree-manip tree init leaf first rest accum)
  (cond ((null? tree) init)
        ((not (pair? tree)) (leaf tree))
        (else (accum
                 (tree-manip (first tree) init leaf first rest accum)
                 (tree-manip (rest tree) init leaf first rest accum)))))
```

write calls to perform the operations described below. Suppose that we provide a test tree,

```
(define test-tree '(1 (2 (3 (4) 5) 6) 7))
```

Write the call to `tree-manip` that will compute the sum of the leaves of `test-tree`, resulting in a return value of 28.

```
(tree-manip _____
             _____
             _____
             _____
             _____
             _____
             _____)
```

Write the call to `tree-manip` that will flatten the `test-tree`, resulting in a return value of `(1 2 3 4 5 6 7)`.

```
(tree-manip _____
             _____
             _____
             _____
             _____
             _____)
```

Problem 5:

On the last page of the quiz, you will find an incomplete environment diagram. Tear out that page and use it as a reference to provide values for all of the entries that appear in angle brackets. The value for each entry should be taken from the object titles to the upper left of each object (e.g., GE, E1, P1, etc.). Note that the sequencing of object titles has no relation to the order in which the objects were created.

Assume the statements below are evaluated in order. Fill in the blanks below to give values for the missing pieces of the environment diagram.

```
(define (f1 x)
  (let ((a (* x x)))
    (lambda (m) (+ a x m))))

(define f2 (f1 3))

(f2 4)

(f2 5)
```

<1>: _____

<2>: _____

<3>: _____

<4>: _____

<5>: _____

<6>: _____

<7>: _____

<8>: _____

<9>: _____

<10>: _____

<11>: _____

Problem 6:

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)))
    (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 6 (continued):

Code is repeated here for your convenience.

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

a) 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.)

Problem 7:

The `map` procedure for lists applies a procedure to each element of a list, building up a new list with the new values. In this problem, you'll write `map!`, which will modify the existing list to have the values of the operator applied to the original values.

For example, if we

```
(define a '(1 2 3 4 5))
```

then call

```
(map! (lambda (x) (* x 2)) a)
```

which returns

```
done
```

Then evaluate `a`, we see that the value of `a` is now:

```
(2 4 6 8 10)
```

Fill in the missing code in the following definition of `map!` to complete the procedure.

```
(define (map! op lst)
  (if (null? lst)
      'done
      <fill in code here>))
```

TEAR OUT THIS PAGE FOR USE WITH PROBLEM 3.

