Sample final exam #1

Problem 1 (Higher order procedures).

Write a procedure `ordinal` that takes a nonnegative integer argument n. It should return a procedure that takes a list as argument and returns the nth element of the list (counting from one):

```
> (define third (ordinal 3))
> (third '(John Paul George Ringo))
George
```

Hint: `list-ref`

Problem 2 (Iterative and recursive processes).

The following recursive procedure takes a list of integers and returns the number of elements that are even.

```
(define (count-evens ints)
  (cond ((null? ints) 0)
        ((even? (car ints)) (+ 1 (count-evens (cdr ints))))
        (else (count-evens (cdr ints)))))
```

Rewrite this as an iterative procedure by filling in the blanks below.

```
(define (count-evens ints)

  (define (helper ____________________________)

    (cond ________________________________________________________

          ________________________________________________________

          ________________________________________________________))

  (helper ________________________________________________________))
```

Problem 3 (Abstract data types).

The following program is designed for use with the "world tree" discussed in lecture, in which the nodes represent geographical regions (contries, states, cities). The selectors for Trees are `datum` and `children`. The datum at each node is a word or sentence. The program, which returns a list of all the cities whose first word is "San," does not respect data abstraction:

```
(define (find-san-cities tree)
  (if (null? (cdr tree))        ; Is this node a city?
      (if (equal? (car (car tree)) 'San)
          (list (car tree))
          '())
      (san-helper (cdr tree))))

(define (san-helper forest)
  (if (null? forest)
      '()
      (append (find-san-cities (car forest))
              (san-helper (cdr forest)))))
```

In the version below, each `car` and `cdr` has been replaced by a blank. Fill in the blanks with the correct selectors, respecting all the relevant abstract data types:

```
(define (find-san-cities tree)

  (if (null? (______________ tree))

      (if (equal? (______________ (______________ tree)) 'San)

          (list (______________ tree))
          '())

      (san-helper (______________ tree))))

(define (san-helper forest)
  (if (null? forest)
      '()

      (append (find-san-cities (______________ forest))

              (san-helper (______________ forest)))))
```

Problem 4 (Trees).

Consider the following program:

```
(define (cost tree)
  (cost-help tree 0))

(define (cost-help tree above)
  (let ((new (+ (datum tree) above)))
    (make-node new
               (map (lambda (child) (cost-help child new))
                    (children tree)))))
```

If `cost` is called with the following Tree as its argument, draw the Tree that it returns.

```
       3
      / \
     /   \
    2     1
   / \   / \
  1   0 5   3
```

Problem 5 (Generic operators).

There's a collection of programs called `netpbm` that is used to convert images from one format to another (for example, to convert gif files to jpeg files). The collection consists of around 160 *converters.* Each converter is a program that knows how to convert a file in one specific format to one other format. So, for example, there's a converter called `tifftopnm` that converts images in tiff format to images in pnm format.

The documentation for `netpbm` claims that the programs can be used to convert to and from any of 80 image formats. How is this possible with only 160 converters?

Your answer must be no more than 30 words!

Problem 6 (Object oriented programming).

Here is a definition in OOP language of the class `line-obj` from project 4:

```
(define-class (line-obj text)
  (method (next)
    (let ((result (car text)))
      (set! text (cdr text))
      result))
  (method (empty?) (null? text))
  (method (put-back token) (set! text (cons token text))) )
```

Write an equivalent program in ordinary Scheme without using the OOP language. Here is an example of how your program will be used:

```
> (define logo-line (make-line-obj '(print 3 print 4 print 5)))
okay
> (logo-line 'next)
print
> (logo-line 'next)
3
> (logo-line 'empty?)
#f
> ((logo-line 'put-back) 'hello)       ;note double parentheses
okay
> (logo-line 'next)
hello
> (logo-line 'next)
print
```

Don't check for errors. Here is the first line of the program, continue from here:

```
(define (make-line-obj text)
```

Problem 7 (Environment diagrams).

Consider the following definitions.

```
(define x 1)

(define (charm x)
  (lambda (y) (+ x y)))

(define z ((charm 3) 5))
```

Below are two partial environment diagrams for these definitions. They are missing two things:

(1) An arrow from the `y --> 5` frame to the environment that it extends;

(2) The value of `z` in the global frame.

Complete the first diagram as it would appear in lexically scoped Scheme; complete the second diagram as it would appear if Scheme used dynamic scope.

```
LEXICAL SCOPE
-------------


 ____global_
|           |
|  x --> 1  |<------------,
|           |           _  |
|  charm ------------>(_)(_)
|           |   params: x
|  z -->___ |   body: (lambda (y)
|           |            (+ x y))
|           |<------------,
|___________|             |
                       ___|_______
                      |           |
           +--------->|  x --> 3  |
           |          |___________|
       (_)(_)
       params: y
       body: (+ x y)   ___________
                      |           |
                      |  y --> 5  |
                      |___________|
```

```
DYNAMIC SCOPE
-------------


 ____global_
|           |
|  x --> 1  |<------------,
|           |           _  |
|  charm ------------>(_)(_)
|           |   params: x
|  z -->___ |   body: (lambda (y)
|           |            (+ x y))
|           |<------------,
|___________|             |
                       ___|_______
                      |           |
           +--------->|  x --> 3  |
           |          |___________|
       (_)(_)
       params: y
       body: (+ x y)   ___________
                      |           |
                      |  y --> 5  |
                      |___________|
```

Problem 8 (Mutation).

Write the procedure `interleave!` that takes two lists (not streams!) as arguments and returns the result of interleaving them. **It must do its job by mutation, without allocating any new pairs!** Example:

```
> (define x (list 'a 'b 'c 'd 'e))
> (define y (list 1 2 3 4 5 6 7 8))
> (define z (interleave! x y))
> z
(a 1 b 2 c 3 d 4 e 5 6 7 8)
> x
(a 1 b 2 c 3 d 4 e 5 6 7 8)
> y
(1 b 2 c 3 d 4 e 5 6 7 8)
```

Note that either list might be shorter than the other; your procedure should handle this situation correctly.

Our solution uses five lines of code; if yours takes more than ten, you're probably not thinking about this properly.

Problem 9 (Concurrency).

Consider the following three procedures and two serializers.

```
(define (incr) (set! x (+ x 1)))
(define (decr) (set! x (- x 1)))
(define (double) (set! x (* x 2)))

(define s (make-serializer))
(define t (make-serializer))
```

In each of the following situations, what outcomes are possible? For each case, choose the single best answer.

```
(define x 0)
(parallel-execute (s (t incr)) (s (t decr)) double)
```

____x is -1, 0, or 1

____x is -1, 0, or 1, or possible deadlock

____x is -2, -1, 0, 1, or 2

____x is -2, -1, 0, 1, or 2, or possible deadlock

```
(define x 0)
(parallel-execute (s incr) (s (t decr)) (s double))
```

____x is -1, 0, or 1

____x is -1, 0, or 1, or possible deadlock

____x is -2, -1, 0, 1, or 2

____x is -2, -1, 0, 1, or 2, or possible deadlock

Problem 10 (MapReduce).

Suppose we've done some MapReduce computations, and now we have `my-pairs`, the following stream of key-value pairs:

```
> (ss my-pairs)
((bar . 2) (bat . 3) (baz . 1) (big . 8) (bill . 7) (bin . 6) (bog . 0))
```

Now, what is the stream returned by each of the following calls to `mapreduce`? If the result is an error, just say ERROR. Otherwise, show the entire stream as `show-stream` does (every element, unless there are more than 10).

(A)

```
(mapreduce (lambda (kvp) (list (make-kv-pair (butlast (kv-key kvp))
                                             (kv-value kvp))))
           +
           0
           my-pairs)
```

(B)

```
(mapreduce (lambda (kvp) (list (make-kv-pair (butlast (kv-key kvp))
                                             (kv-value kvp))))
           (lambda (x y) (+ (kv-value x) (kv-value y)))
           0
           my-pairs)
```

Problem 11 (Streams).

Given the following input, how will the Scheme interpreter respond? Fill in the blanks in the transcript below.

(Assume the definition of show-stream used in the lectures, which displays the first few elements of a stream.)

```
> (define (spew x) (cons-stream x (spew x)))
spew
> (define garply (cons-stream 1 (stream-map + (spew 1) garply)))
garply
> (show-stream garply 10)
```

____

```
> (define strange (cons-stream '() (stream-map cons garply strange)))
strange
> (show-stream strange 5)
```

____

Problem 12 (Lazy evaluator).

Suppose that the following expressions are entered into the **lazy** interpreter in the order shown.

```
;;; L-Eval input:
(define (truth x y) (display (+ x 1)) y)
                                                <----- point A
;;; L-Eval input:
(define beauty (truth (* 6 7) (- 5 2)))
                                                <----- point B
;;; L-Eval input:
beauty
                                                <----- point C
```

The three primitives `+`, `*`, and `-` will each be executed exactly once at some point during the session. Indicate when each of them will occur.

`+` will be executed at point ____

`*` will be executed at point ____

`-` will be executed at point ____

Problem 13 (Nondeterministic evaluator).

Here is a program written for the non-deterministic evaluator:

```
(define b 1)

(define (clue)
  (let ((a (amb 1 2 3 4 5)))
    (require (= 0 (remainder a 2)))
    (set! b (+ b a))
    b))
```

Fill in the results from the non-determinisitic evaluator:

```
Amb-Eval input:   (clue)

Amb-Eval value:   ____________________________

Amb-Eval input:   try-again

Amb-Eval value:   ____________________________
```

Problem 14 (Nondeterministic evaluator).

Here is a program written for the non-deterministic evaluator:

```
(define (mystery)
  (let ((a (an-integer-starting-at 1))
        (b (an-integer-starting-at 1)))
    (require (= b (* a a)))
    (list a 'squared '= b)))

(define (an-integer-starting-at n)
  (amb n (an-integer-starting-at (+ n 1))))
```

Fill in the results from the non-deterministic evaluator:

```
Amb-Eval input:   (mystery)

Amb-Eval value:   ____________________________

Amb-Eval input:   try-again

Amb-Eval value:   ____________________________
```

Problem 15 (Logic programming).

This is a question about logic programming. In it, as in some of the sample exams, we represent nonnegative integers by lists containing the letter `a` repeatedly; for example, `(a a a)` represents the number 3.

Don't worry about your rule(s) working backward; only the last element of the query will be an unbound variable. Don't use `lisp-value` in your solution.

Write a logic program to find the depth of a list, equivalent to this Scheme procedure:

```
(define (depth lst)
  (if (not (pair? lst))
      0
      (max (+ 1 (depth (car lst)))
           (depth (cdr lst)))))
```

```
Query input:   (depth (a (b c (d) e) f) ?x)
Query result:  (depth (a (b c (d) e) f) (a a a))
```

We provide the logic equivalent of `pair?` and `max`:

```
(assert! (rule (pair ?x)
               (same ?x (?p . ?q))))

(assert! (max () () ()))

(assert! (rule (max () (a . ?x) (a . ?x))))

(assert! (rule (max (a . ?x) () (a . ?x))))

(assert! (rule (max (a . ?x) (a . ?y) (a . ?z))
               (max ?x ?y ?z)))
```

Here's an example of how `max` is used:

```
Query input:   (max (a a a a a) (a a a) ?x)
Query result:  (max (a a a a a) (a a a) (a a a a a))
```

Problem 16 (Metacircular evaluator).

You are to modify the metacircular evaluator to allow *optional arguments* to a procedure. The formal parameter list for a defined procedure may contain sublists of the form

```
(parameter-name default-value)
```

to indicate that this parameter is optional. All optional parameters must come after all required parameters in the list. When the procedure is invoked, if not enough actual arguments are supplied, the default values are used for the extra parameters. Here is an example. Suppose we say

```
(define (foo a b (c 3) (d 4))
   ...body...)
```

Here are possible invocations:

```
invocation                    a   b   c   d

(foo 50 60 70 80)            50  60  70  80
(foo 50 60 70)               50  60  70   4
(foo 50 60)                  50  60   3   4
(foo 50)                   ERROR, too few args
(foo 50 60 70 80 90)       ERROR, too many args
```

You may assume that the default value will always be a number (therefore self-evaluating) as in this example. Also, you need not check for syntax errors in the formal parameter list of a definition.

For your convenience, part of the metacircular evaluator is reproduced on the two following pages of this exam. **You can solve this problem by modifying only one procedure** from among the ones listed, and perhaps writing new helper procedures for it.

Hint 1: Do you have to change the process of defining a procedure, or of calling a procedure?

Hint 2: Is this mainly a change to `eval` (about expressions) or to `apply` (about calling procedures)?

[For reproduction in the reader, we omit the evaluator code to avoid repetition.]