**Sample final exam #3**

**Problem 1 (Domain and range).**

Describe clearly but briefly **(five words or less)** the domain and range of each of the following procedures:

(a) The `children` procedure for Trees

(b) The following procedure from the adventure game:

```
(define (owner thing)
  (let ((obj (ask thing 'possessor)))
    (if (eq? obj 'no-one)
        'no-one
        (ask obj 'name) )))
```

(c) `set-cdr!`

**Problem 2 (Data abstraction).**

The following function takes as its argument a list of rational numbers, and returns the product of all the numbers in the list. As in the text, a rational number is a pair `(numer . denom)`. The function should use the selectors `numer` and `denom`, and the constructor `make-rat`, but it doesn't:

```
(define (ratprod lst)

  (define (helper lst n d)

    (if (null? lst)

        (cons n d)

        (helper (cdr lst) (* n (caar lst)) (* d (cdar lst)))))

  (helper lst 1 1))
```

Modify `ratprod` (by crossing out and inserting above) to use data abstraction correctly.

**Problem 3 (Orders of growth).**

(a) The function below computes powers of 2 recursively. For example, `(two-to-the 8)` is 256.

```
(define (two-to-the n)
  (if (zero? n)
      1
      (let ((prev (two-to-the (- n 1))))
        (* prev 2) )))
```

In terms of big-theta notation, how long does `(two-to-the n)` take to run?

$\Theta$(____)

(b) Suppose we change the function to look like this:

```
(define (two-to-the n)
  (if (zero? n)
      1
      (let ((prev (two-to-the (- n 1))))
        (+ prev prev) )))           ; <-- changed code in boldface
```

How long does the revised function take to run?

$\Theta$(____)

(c) Suppose we change the function again to look like this:

```
(define (two-to-the n)
  (if (zero? n)
      1
      (+ (two-to-the (- n 1))
         (two-to-the (- n 1)) )))
```

How long does the revised revised function take to run?

$\Theta$(____)

**Problem 4 (Higher order functions).**

Here are some useful functions of functions:

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (specialize f val)
  (lambda (x) (f val x)))

(define (swap-args f)
  (lambda (x y) (f y x)))
```

And here are some examples of how to use them:

```
(define double (specialize * 2))
(define second (compose first bf))
(define vowel? (specialize (swap-args member?) 'aeiou))
```

In the examples below, you're shown sample interactions with functions `initials`, `positive?`, and `leaf?`, followed by proposed definitions for each. Pick the correct definition.

```
> (initials '(she loves you))
(s l y)

____ (define initials (specialize first every))
____ (define initials (specialize every first))
____ (define initials (compose every first))
____ (define initials (compose first every))
```

```
> (positive? 5)
#t
> (positive? -3)
#f

____ (define positive? (specialize > 0))
____ (define positive? (compose > 0))
____ (define positive? (specialize (swap-args >) 0))
____ (define positive? (compose (swap-args >) 0))
```

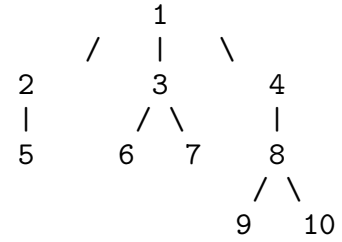
```
> (leaf? world-tree)
#f
> (leaf? (make-tree 7 '()))
#t

____ (define leaf? (compose null? children))
____ (define leaf? (compose children null?))
____ (define leaf? (specialize null? children))
____ (define leaf? (specialize children null?))
```

**Problem 5 (Trees).**

The following function operates on the Tree abstract data type as discussed in lecture:

```
(define (mystery tree)
  (if (null? (children tree))
      (make-tree (datum tree) '())
      (make-tree (+ (datum tree) 100)
                 (map mystery (cdr (children tree))))))
```

Draw the result of calling `mystery` with the tree below as its argument:




**Problem 6 (Object oriented programming).**

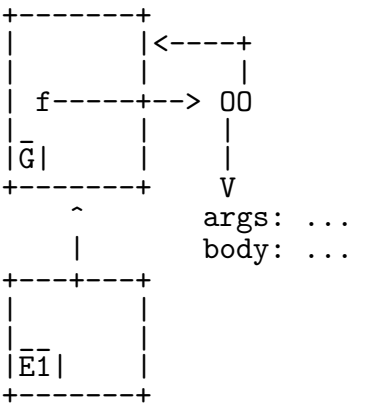
Write an OOP-notation class definition equivalent to the following:

```
(define foo
  (let ((a 3))
    (lambda (b)
      (let ((c 4))
        (lambda (message)
          (cond ((eq? message 'hi)
                 (+ a b))
                (else
                 (+ c message))))))))
```

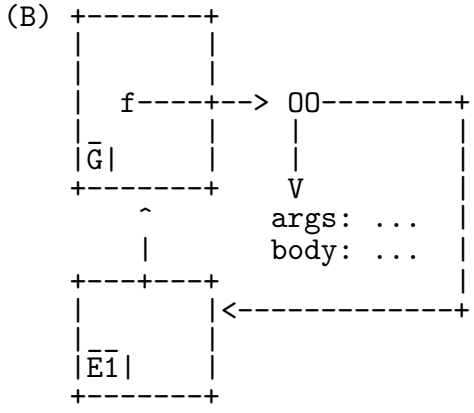
**Problem 7 (Environment diagrams).**

Below are four environment diagrams and four pieces of Scheme code. Match each environment diagram to the Scheme expression which generates it. **Ignore the implicit-lambda bubbles produced by `let` expressions.**

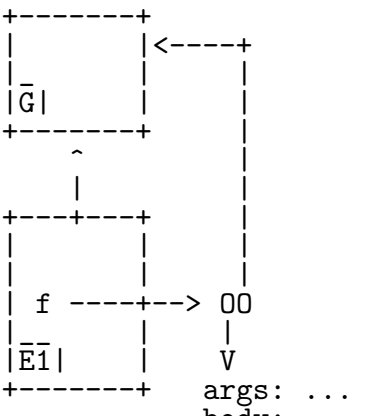
(A)



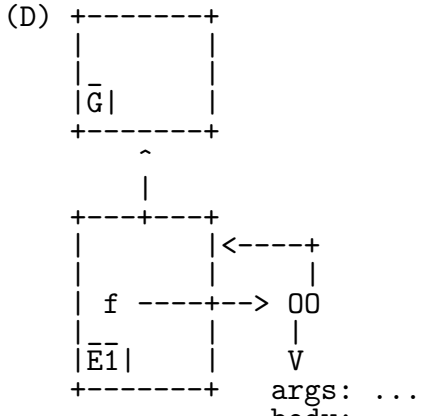

(B)




(C)




(D)




```
(let ((f (lambda (x) x)))
 'okay)
```

_____

```
(define f
 (let ()
   (lambda (x) x)))
```

_____

```
(let ()
 (define f (lambda (x) x)))
```

_____

```
(define (f) f)
(f)
```

_____

**Problem 8 (Message passing).**

We've seen that we can define a pair as a function which returns its `car` or `cdr` on request, like this:

```
(define (make-pair x y)
  (lambda (msg)
    (cond ((eq? msg 'car) x)
          ((eq? msg 'cdr) y)
          (else (error "I have no idea what you're talking about.")) )))
```

We can use more than one of these pairs, linked in the usual way, to make a list:

```
> (define abc (make-pair 'a (make-pair 'b (make-pair 'c '() ))))
> (abc 'car)
a
```

It would be nice if we could use abbreviated messages like `cadr` as we can for ordinary lists:

```
> (abc 'cadr)
b
```

In practice we'd add messages for all possible combinations (`caar`, `cadr`, `cdar`, `cddr`, etc.), but for the purposes of this exam, **you only have to add the `cadr` message.** You don't have to worry about handling the error of sending this message to a pair that doesn't have a cadr. Finish the definition here:

```
(define (make-pair x y)
  (lambda (msg)
    (cond ((eq? msg 'car) x)
          ((eq? msg 'cdr) y)
```

**Problem 9 (Mutation).**

We type the following into Scheme:

```
(define lst (list (list 'a 'b) (list 'c 'd) 'e))
(set-car! (cadr lst) (cddr lst))
(set-car! (car lst) (caddr lst))
(set-cdr! (car lst) (cdar lst))
(set-car! (cddr lst) 'x)
```

Show the final box and pointer diagram and the printed value of `lst`.

**Problem 10 (Metacircular evaluator and Logo).**

This question is about the metacircular evaluator and the Logo interpreter. In both cases, assume that the evaluator has been initialized (by calling `mce` or `initialize-logo`), but *you are now talking to the underlying STk prompt.*

What would STk print in response to:

```
STk> (mc-eval '(+ a 3) (list (make-frame '(a) '(3))))
```

---

How about:

```
STk> (logo-eval (make-line-obj '(sum 2 :a)) (list (make-frame '(a) '(3))))
```

---

**Problem 11 (Streams).**

What are the first 10 elements of the following stream:

```
(define foo
  (cons-stream 'a
               (cons-stream 'b
                            (interleave foo
                                        (stream-filter (lambda (x)
                                                         (eq? x 'b))
                                                       foo)))))
```

___ ___ ___ ___ ___ ___ ___ ___ ___ ___

**Problem 12 (Logic programming).**

Write logic rules for the `assoc` relation, as in these examples:

```
QUERY:  (assoc  b  ((a . 3) (b . 4) (c . 5) (b . 6))  ?x)
RESULT: (assoc  b  ((a . 3) (b . 4) (c . 5) (b . 6))  (b . 4))

QUERY:  (assoc  ?w ((a . 3) (b . 4) (c . 5) (b . 6))  ?x)
RESULT: (assoc  a  ((a . 3) (b . 4) (c . 5) (b . 6))  (a . 3))
RESULT: (assoc  b  ((a . 3) (b . 4) (c . 5) (b . 6))  (b . 4))
RESULT: (assoc  c  ((a . 3) (b . 4) (c . 5) (b . 6))  (c . 5))

QUERY:  (assoc  g  ((a . 3) (b . 4) (c . 5) (b . 6))  ?x)
```
(No results match this one.)

Note that only the first matching key in the association list counts!

**Do not use `lisp-value`!**

**Problem 13 (Concurrency).**

Given the following definitions:

```
(define x 10)
(define (f) (set! x (+ x 3)))
(define (g) (set! x (* x 2)))
(define s (make-serializer))
(define t (make-serializer))
```

what are the possible results of each of the following:

```
(parallel-execute (s f) (t g))

____ can produce incorrect results
____ can deadlock
____ both
____ neither
```

**(Continued on next page.)**

```
(parallel-execute (s f) (s g))

____ can produce incorrect results
____ can deadlock
____ both
____ neither


(parallel-execute (s (t f)) (t g))

____ can produce incorrect results
____ can deadlock
____ both
____ neither


(parallel-execute (s (t f)) (s g))

____ can produce incorrect results
____ can deadlock
____ both
____ neither


(parallel-execute (s (t f)) (t (s g)))

____ can produce incorrect results
____ can deadlock
____ both
____ neither
```

**Problem 14 (Trees).**

You are given the Tree ADT with constructor `make-tree` and selectors `datum` and `children`, and a Binary Tree ADT with constructor `make-bt` and selectors `value`, `left-branch`, and `right-branch`. (The constructor is named differently from the one in the text to make sure there's no confusion with the constructor for trees.)

Write the procedure `tree->bt` that takes a Tree as its argument. If any node in the tree has more than two children, `tree->bt` should return `#f`. Otherwise, it should return a Binary Tree with the same data and the same shape as the argument. (If a node in the Tree has one child, it should appear as the left child of the corresponding node in the Binary Tree.)

**Problem 15 (Mutation).**

Write the procedure `swap!` that takes a list as its argument, and returns the same list with pairs of elements interchanged, like this:

```
> (swap! (list 'a 'b 'c 'd 'e 'f))     ; even number of elements
(b a d c f e)
> (swap! (list 'a 'b 'c 'd 'e))        ; odd number of elements
(b a d c e)
```

The list should be modified by mutation; **do not allocate any new pairs.**

**Problem 16 (Metacircular evaluator).**

Suppose you've defined a function with more than one argument, such as

```
(define (fast-exp base exp)
  (cond ((= exp 0) 1)
        ((even? exp) (fast-exp (* base base) (/ exp 2)))
        (else (* base (fast-exp base (- exp 1))))))
```

A common error made by students who are just learning about higher order procedures is to try to use a function like `fast-exp` this way:

```
(map (fast-exp 2) '(4 5 7 8))        ; wrong!
```

instead of the correct

```
> (map (lambda (exp) (fast-exp 2 exp)) '(4 5 7 8))
(16 32 128 256)
```

Some programming languages accept calls with too few arguments like

```
(fast-exp 2)
```

and interpret them as requests to create specialized procedures — in this example, the same procedure that would be created by

```
(lambda (exp) (fast-exp 2 exp))
```

Add this feature to the metacircular evaluator. Whenever a call to a compound procedure is given too few arguments to match the procedure's formal parameters, the evaluator should match the procedure's first parameters to the given arguments, and create and return a new procedure that takes only the unmatched parameters. (To make this feature really practical, it would have to work for calls to primitive procedures also, but that's too hard for an exam question.)

**[In the reader we are omitting the code from the metacircular evaluator that was included in the actual exam.]**