MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2005

Final Exam

Closed Book – three sheets of notes

Throughout this quiz, we have set aside space in which you should write your answers. Please try to put all of your answers in the designated spaces, as we will look only in this spaces when grading.

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

NAME:

Section Number: Tutor's Name:

PART	Value	Grade	PART	Value	Grade
1	10		6	25	
2	18		7	22	
3	23		8	20	
4	20		9	28	
5	24		10	10	
Total	200				

Section	Time	Location	Rec. Instructor	Tutors
10H	10:00	34-301	Horn	Dalley
10C	10:00	34-304	Cutler	Yu
10G	10:00	36-144	Koile	various
11G	11:00	26-210	Koile	various
11H	11:00	34-301	Horn	Gleyzer
11C	11:00	26-314	Collins	Danaher
12C	12:00	34-301	Collins	Pritchard
12B	12:00	26-314	Barzilay	Wilson
1B	1:00	34-303	Barzilay	various
1D	1:00	34-301	Durand	Vlasic
2C	2:00	26-322	Cutler	Zhou
2D	2:00	34-303	Durand	various

Part 1 (10 points): Objects with state

It is sometimes convenient to keep track of the arguments with which a procedure has been called. The following procedure keeps track of this for any procedure of one numerical argument:

```
(define (history proc)
  (let ((calls '()))
    (lambda (arg)
      (cond ((number? arg)
             INSERT1)
            ((eq? arg 'calls)
             INSERT2)
            ((eq? arg 'reset)
             INSERT3)
            (else (error "illegal argument"))))))
```

Here are some example usages:

```
(define square
  (history (lambda (x) (* x x))))

(square 'calls)
;Value: #f

(square 5)
;Value: 25

(square 10)
;Value: 100

(square 'calls)
;Value: (10 5)

(square 'reset)
;Value: done

(square 'calls)
;Value: #f
```

Question 1 What code should be provided in place of INSERT1, to produce the above behavior?

Question 2 What code should be provided in place of INSERT2, to produce the above behavior (i.e. the list of supplied arguments is returned)?

Question 3 What code should be provided in place of INSERT3, to produce the above behavior (i.e. the list of supplied arguments is reset to the empty list, and the symbol `done` is returned)?

Part 2 (18 points): Orders of growth

Suppose you have a set of integers, represented as a list. You want to sort the list in increasing order. One way to do this is to use the following idea:

- First split the list into two roughly equal parts
- Sort each sublist, recursively using the same idea
- Then merge the two sorted sublists into a single sorted list

Here are some pieces of code to accomplish this:

```
(define (split lst)
  (define (split-help lst1 lst2 left)
    (if (null? left)
        (list lst1 lst2)
        (split-help lst2 (cons (car left) lst1) (cdr left))))  ; note reversal
  (split-help '() '() lst))

(define (merge lst1 lst2)
  (cond ((null? lst1) lst2)
        ((null? lst2) lst1)
        ((< (car lst1) (car lst2))
         (cons (car lst1) (merge (cdr lst1) lst2)))
        (else (cons (car lst2) (merge lst1 (cdr lst2))))))
```

For each of these procedures, indicate the order of growth of the procedure in using two different resources: the time required, measured by the number of primitive operations performed; and the space required, measured by the maximum number of deferred operations that must accumulated. In each case, this should be measured in terms of the number of elements of the lists given as argument.

Select your answer from the following:

1. constant
2. linear
3. quadratic
4. exponential
5. logarithmic
6. none of the above

Question 4

For the procedure `split`, what are the orders of growth?

Time: [ ] Space: [ ]

Question 5

For the procedure `merge`, what are the orders of growth?

Time: [ ] Space: [ ]

Question 6

Finally, here are some possible procedures for implementing sorting. For each, select from the following list to indicate what happens if we use this procedure:

1. correctly sorts a list in increasing order
2. enters an infinite loop
3. returns a copy of the original list unsorted
4. only sorts the first two elements, the order of the remainder of the list is unchanged
5. something else

```
(define (split-merge-sort lst)
  (cond ((null? lst) '())
        (else (let ((sp (split lst)))
                (merge (split-merge-sort (car sp))
                       (split-merge-sort (cadr sp)))))))
```

[ ]

```
(define (split-merge-sort lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (else (let ((sp (split lst)))
                (merge (split-merge-sort (car sp))
                       (split-merge-sort (cadr sp)))))))
```

[ ]

```
(define (split-merge-sort lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (else (let ((sp (split lst)))
                (merge (split-merge-sort (cadr sp))  ; NOTE: reversed from above
                       (split-merge-sort (car sp)))))))
```

[ ]

Part 3 (23 points): Streams and Lazy Evaluation:

Question 7

Assume that the following have been evaluated:

```
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
   (cons-stream (+ (stream-car s1) (stream-car s2))
                (add-streams (stream-cdr s1) (stream-cdr s2))))
```

Consider the expression

```
(define integers (add-streams ones integers))
```

For each of the following, mark the box if the statement applies to the above scenario:

- ☐ The expression evaluates to a stream of integers;
- ☐ The interpreter goes into an infinite loop when `(stream-cdr integers)` is evaluated;
- ☐ An "unbound variable" error occurs when the above expression defining integers is evaluated;
- ☐ An "unbound variable" error occurs when `(stream-cdr integers)` is evaluated;

Question 8

Assume that the following expression is evaluated:

```
(define fibonaccis
  (cons-stream 1
               (add-streams fibonaccis (stream-cdr fibonaccis))))
```

For each of the following, mark the box if the statement applies to the above scenario:

- ☐ The expression evaluates to a stream of Fibonacci numbers;
- ☐ The interpreter goes into an infinite loop when `(stream-cdr fibonaccis)` is evaluated;
- ☐ An "unbound variable" error occurs when the above expression defining fibonaccis is evaluated;
- ☐ An "unbound variable" error occurs when `(stream-cdr fibonaccis)` is evaluated;

Question 9

Assume that `multiply-streams` has been defined by copying `add-streams` and simply replacing the '+' with a '*', and consider the expression

```
(define mystery
   (cons-stream 1
             (multiply-streams (add-streams ones ones)
                               mystery)))
```

What are the first five elements of `mystery`?

Question 10

Sometimes it is convenient to convert a list to a stream. Fill in the missing part in

```
(define (list->stream lst)
   (if (null? lst)
       the-empty-stream
       YOUR-CODE-GOES-HERE))
```

Question 11

Assume that

```
(define ints
   (cons-stream 1
      (stream-map (lambda (x) (+ x 1)) ints)))
```

To what will the following evaluate?

`(stream-null? (stream-filter odd? (stream-filter even? ints)))`

Choose from:

1. #t
2. #f
3. the number 1
4. none of the above

Question 12:

Consider the following code:

```
(define (initialized-list f n)
   (define (helper n lst)
     (if (= n 0) lst
        (helper (- n 1) (cons (f n) lst))))
   (helper n #f))

;Example output:
(initialized-list (lambda (x) (* x x)) 5)
;Value: (1 4 9 16 25)

(define (accum)
   (let ((count 0))
     (lambda (x)
       (set! count (+ x count))
       count)))
```

What is the value of the statement

```
(initialized-list (accum) 5)
```

in the ordinary (applicative order) evaluator?

What about in the lazy, non-memoized evaluator?

Part 4 (20 points): Syntactic transformations:

When we examined the Meta-Circular evaluator, we saw that one way to add new forms to our language was to create syntactic transformations, in which a special form was converted, using manipulation of the syntactic representation (e.g., the list structure of the expression), into a form that the evaluator could already handle. For example, a `cond` can be transformed into a nested set of `if` expressions.

We're now going to implement `list` within the evaluator, using syntactic sugar. Recall that `list` statements are equivalent to a sequence of nested `cons` statements, for example

```
(list 1 2 3 4)
```

is equivalent to

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

Note: as in regular scheme we assume that the expression `(list)` evaluates to `nil`, i.e., the empty list.

Suppose we add the following clause to `m-eval`:

```
((list?  exp) (m-eval (list->cons (list-body exp)) env))
```

where

```
(define (list? exp) (tagged-list? exp 'list))
(define (list-body exp) (cdr exp))
```

We implement `list->cons` as follows:

```
(define (list->cons exp)
        ANSWER-13)
```

Question 13

Complete the code for ANSWER-13.

We're now going to implement a similar procedure, `cons*`, using syntactic sugar. Expressions involving `cons*` take the following form:

`(cons* exp-1 exp-2 exp-3 ... exp-n-1 exp-n)`

they are equivalent to the following:

`(cons exp-1 (cons exp-2 (cons exp-3 ... (cons exp-n-1 exp-n))))` For example,

`(cons* 1 2 3)`

is equivalent to

`(cons 1 (cons 2 3))`

In addition, we assume that `cons*` always takes at least one argument, for example `(cons*)` is not allowed. In the case of just one argument, `(cons* exp-1)` is equivalent to exp-1. For example,

`(cons* 3)`

is equivalent to

`3`

Suppose we add the following clause to `m-eval`:

`((cons*? exp) (m-eval (cons*->cons (cons*-body exp)) env))`

where

```
(define (cons*? exp) (tagged-list? exp 'cons*))
(define (cons*-body exp) (cdr exp))
```

We implement `cons*->cons` as follows:

```
(define (cons*->cons exp)
        ANSWER-14)
```

Question 14

Complete the code for ANSWER-14.

Part 5 (24 points): Meta-circular Evaluator:

We are going to add a new special form to the Meta-Circular evaluator (the code for which is included at the end of the exam). This form is a `repeat` expression, examples of which are:

```
(define i 5)
(repeat 4 (display i) (set! i (+ i 1)))

5
6
7
8
;Value: done


(define i 5)
(repeat (+ 1 2) (display i) (set! i (+ i 1)))

5
6
7
;Value: done
```

The syntax of a repeat statement is as follows. The first clause is an expression (e.g. `4` or `(+ 1 2)` which specifies the number of times that the "body" of the statement should be evaluated. The remaining clauses (e.g., `(display i) (set! i (+ i 1))`) are a sequence of clauses which form the "body" of the repeat clause, and should be repeated a number of times. The final value of the repeat clause is the symbol `done`.

Assume we have the following syntactic operators on `repeat` clauses:

```
(define (repeat-first-clause exp) (cadr exp))
(define (repeat-body exp) (cddr exp))
```

To implement the special form, we add a dispatch clause to `m-eval`, and create a new evaluation procedure:

```
(define (m-eval exp env)
   (cond ...
       ((repeat? exp)
            (eval-repeat exp env)))
         ...
       ((application? exp) ...)
       (else ...))

(define (eval-repeat exp env)
        (eval-repeat-doit (m-eval (repeat-first-clause exp) env)
                          (repeat-body exp)
                          env))
```

```
(define (eval-repeat-doit number body env)
        (if (= number 0)  ;test to see if we're at the final test
            ANSWER-15 ;return the final value
            (begin ANSWER-16 ;evaluate the body
                   ANSWER-17))) ;go to next iteration
```

Question 15 Provide an expression for ANSWER-15

Question 16 Provide an expression for ANSWER-16

Question 17 Provide an expression for ANSWER-17

We're now going to implement a similar special form `newrepeat`, which has slightly different behavior. `newrepeat` has the same syntax as `repeat`, for example

```
(define i 5)
(newrepeat 4 (display i) (set! i (+ i 1)))
```

However it has the following behavior: again, it will evaluate the body of the expression up to `n` times, where `n` is the value of the first expression. However, if at any point the body evaluates to the value #f, the `newrepeat` loop exits. Again, `newrepeat` always returns the symbol `done`. Some examples:

```
(define i 5)
(newrepeat 4 (display i) (set! i (+ i 1)) #t)

5
6
7
8
;Value: done

(define i 5)
(newrepeat 4 (display i) (set! i (+ i 1)) #f)

5
;Value: done

(define i 5)
(newrepeat 4 (display i) (set! i (+ i 1)) (< i 6))

5
6
7
;Value: done
```

Here's some new code that we add to the evaluator:

```
(define (m-eval exp env)
   (cond ...
       ((newrepeat? exp)
            (eval-newrepeat exp env)))
         ...
       ((application? exp) ...)
       (else ...))

(define (eval-newrepeat exp env)
        (eval-newrepeat-doit (m-eval (repeat-first-clause exp) env)
                             (repeat-body exp)
                             env))
```

```
(define (eval-newrepeat-doit number body env)
        (if (= number 0)  ;test to see if we're at the final test
            ANSWER-18 ;return the final value
            ANSWER-19)) ;recurse unless the body evaluates to #f
```

Question 18 Provide an expression for ANSWER-18

Question 19 Provide an expression for ANSWER-19

Finally, we'd like to add some error checking to our implementation of `newrepeat`. In particular, if the first clause of `newrepeat` does not have a value which is a number, then the evaluator should return the symbol `error`. Otherwise, it should behave as before. For example:

```
(define i 5)
(newrepeat 4 (display i) (set! i (+ i 1)) #f)

5
;Value: done

(define i 5)
(newrepeat 'test (display i) (set! i (+ i 1)) #f)

;Value: error
```

Question 20 What is your solution for ANSWER-20 in the following piece of code?

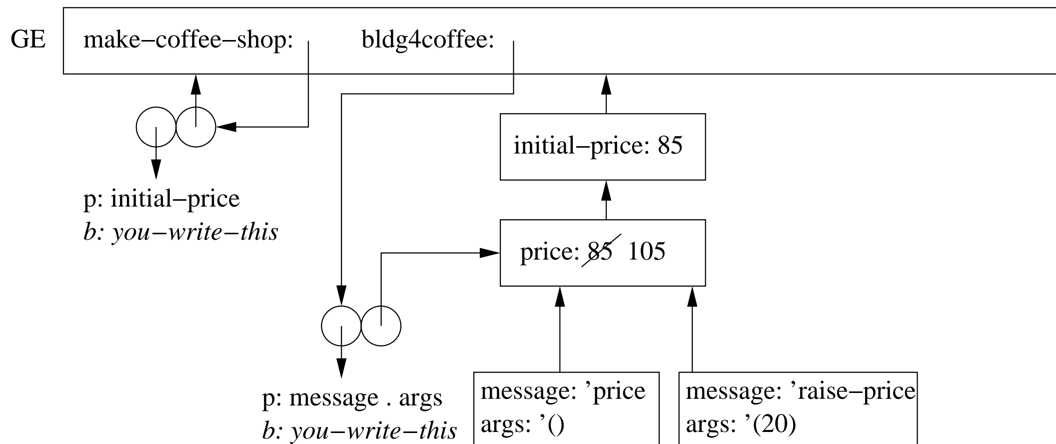
```
(define (eval-newrepeat exp env)
      ANSWER-20)
```

Part 6 (25 points): Environment Diagrams:

Alyssa P. Hacker has been hired by the Building 4 Coffee Shop to help with their new discount program for regular customers. First she tackles the problem of storing the price for a cup of coffee as illustrated in this message-passing style interaction proposed by the managers.

```
(define bldg4coffee (make-coffee-shop 85))
(bldg4coffee 'price) ==> 85
(bldg4coffee 'raise-price 20)  ==> done
(bldg4coffee 'price) ==> 105
```

Here is a partially completed environment diagram drawn by Alyssa.



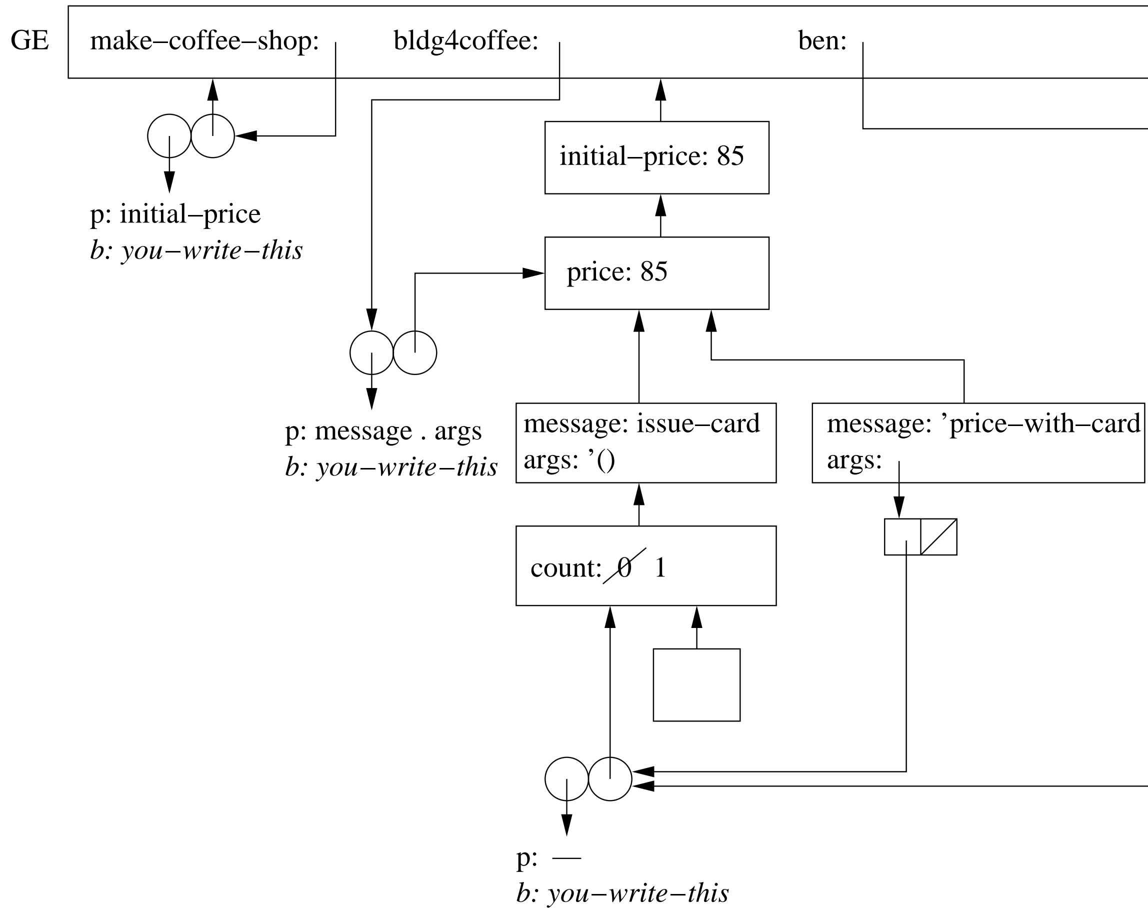
Question 21 Finish the code for `make-coffee-shop` that will result in the environment diagram based on these evaluations. Assume that the only messages handled by a "coffee shop" are shown in the examples.

```
(define (make-coffee-shop initial-price)
  YOUR-CODE)
```

Next Alyssa needs to implement the discount card program: buy 3 cups of coffee at the regular price, get the 4th cup free! To implement this, the coffee shop object will need to handle two additional messages, `issue-card` and `price-with-card`, as illustrated below:

```
(define bldg4coffee (make-coffee-shop 85))
(bldg4coffee 'price) ==> 85
(define ben (bldg4coffee 'issue-card))
(define louis (bldg4coffee 'issue-card))
(bldg4coffee 'price-with-card ben) ==> 85
(bldg4coffee 'price-with-card louis) ==> 85
(bldg4coffee 'price-with-card ben) ==> 85
(bldg4coffee 'price-with-card ben) ==> 85
(bldg4coffee 'price-with-card ben) ==> 0
(bldg4coffee 'raise-price 20) ==> done
(bldg4coffee 'price) ==> 105
(bldg4coffee 'price-with-card louis) ==> 105
(bldg4coffee 'price-with-card louis) ==> 105
(bldg4coffee 'price-with-card louis) ==> 0
```

Here is a partially completed environment diagram drawn by Alyssa.



Question 22 Finish the code for `make-coffee-shop` that will result in this environment diagram.

```
(define (make-coffee-shop initial-price)
  YOURCODE)
```

Part 7 (22 points): Higher Order Procedures:

Question 23 Write a procedure `steps-to-zero`, which takes as argument a numeric procedure `f`, and returns a procedure `g`, such that `g(n)` is the minimum number of `f` steps that result in a zero when starting with `n`. More precisely, if we define

```
(define g (steps-to-zero f))
```

then `(g n)` would return the value `m` if and only if

```
  (f (f (f ... (f n) ...))) = 0
   \____________/
         m
```

For example:

```
(define (collatz n)
  (cond ((= n 1) 0)
        ((even? n) (/ n 2))
        (else (+ 1 (* n 3)))))

(define collatz-steps (steps-to-zero collatz))

(collatz-steps 0) --> 0

(map collatz-steps '(0 1 2 3 4 5 6 7 8 9 10))
    --> (0 1 2 8 3 6 9 17 4 20 7)
```

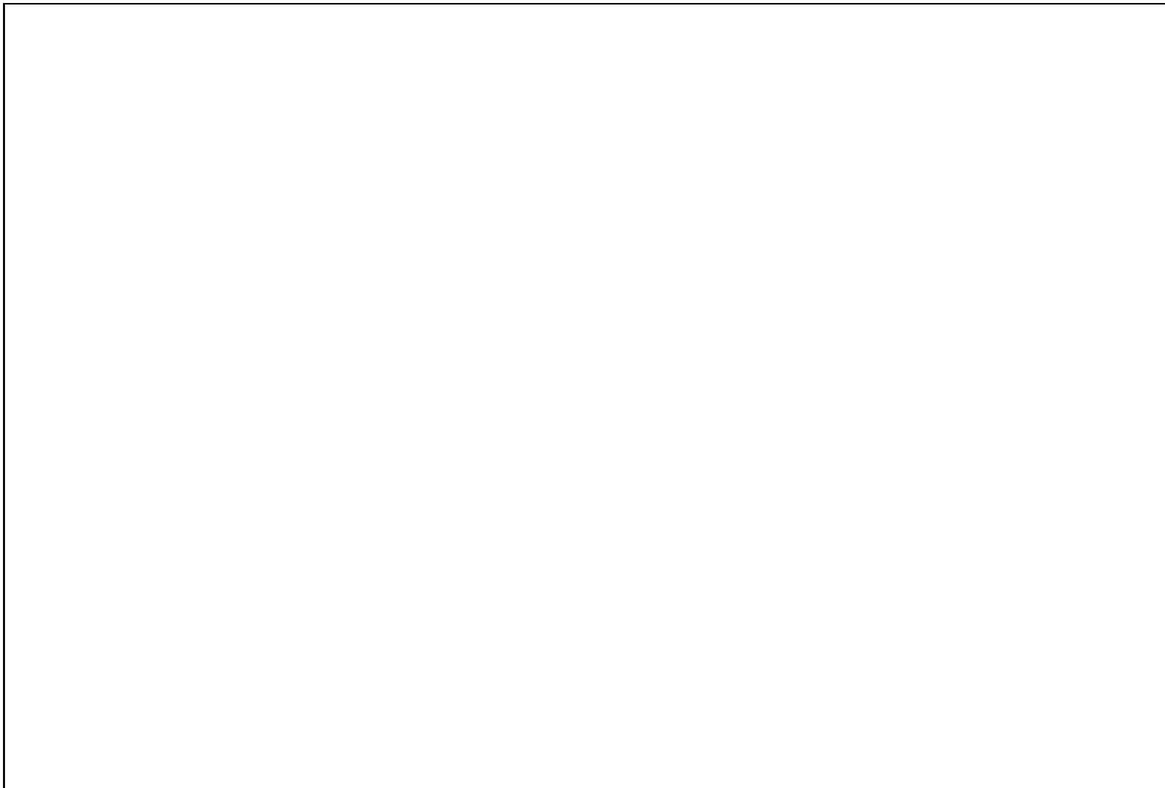
In other words, `collatz-steps` tells us if we apply `collatz` to some argument, then apply it again to the result, and so on, how many applications are needed until the result is 0. Although it might seem like there is a possibility that this process enters an infinite loop (for some `f` and some initial argument), we are going to ignore this for the purposes of this question.

Write the procedure `steps-to-zero`. Hint: think about what type of value this procedure should return. Hint: you may find it useful to think about an auxiliary procedure that keeps track of the number of applications of the argument procedure and the current value after those applications.

Question 24 Write `min-steps-to-zero` which is an extension of `steps-to-zero`: it takes as input a **list** of procedures `f1,...fi`, and returns a procedure `g` such that `g(n)` is the minimum of `steps-to-zero(f1)(n), ..., steps-to-zero(fi)(n)`, that is, it is the minimum of `steps-to-zero` of each procedure in the list, each applied to `n`. Again, don't worry about infinite loops.

Example:

```
(define foo
  (min-steps-to-zero (list collatz (lambda (n) (- n 1)))))

(map foo '(0 1 2 3 4 5 6 7 8 9))
    --> (0 1 2 3 3 5 6 7 4 9)
```

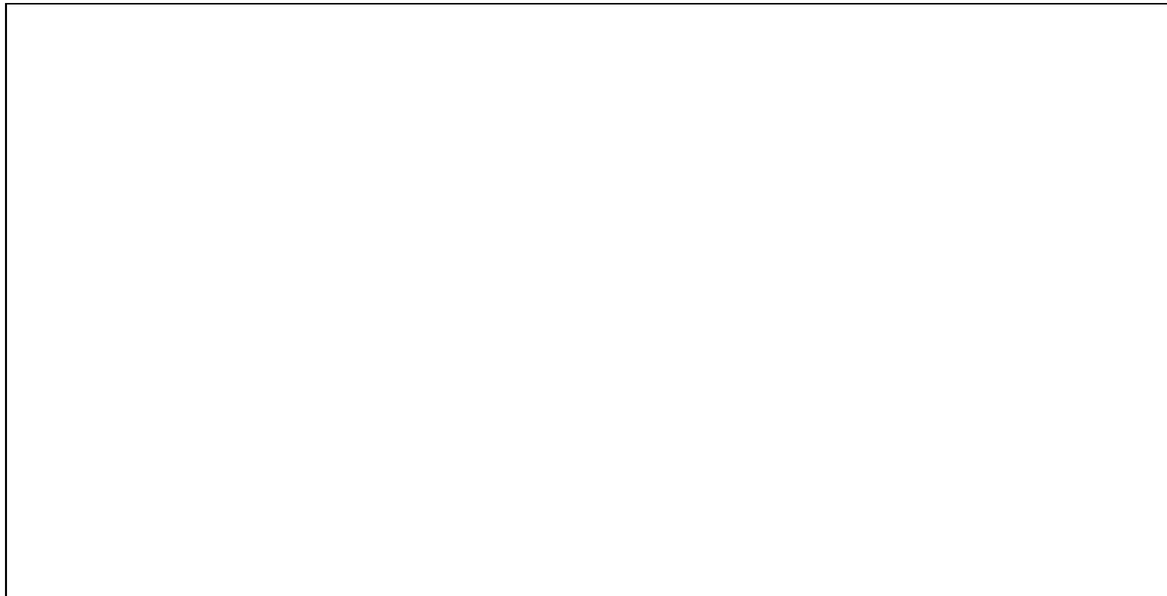


Part 8 (20 points): List structures:

Question 25 Define a `subsets` procedure that returns all subsets of a given list.

For example,

```
(subsets '(1 2))
   --> ((1 2) (1) (2) ())

(subsets '(1 2 3))
  --> ((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```

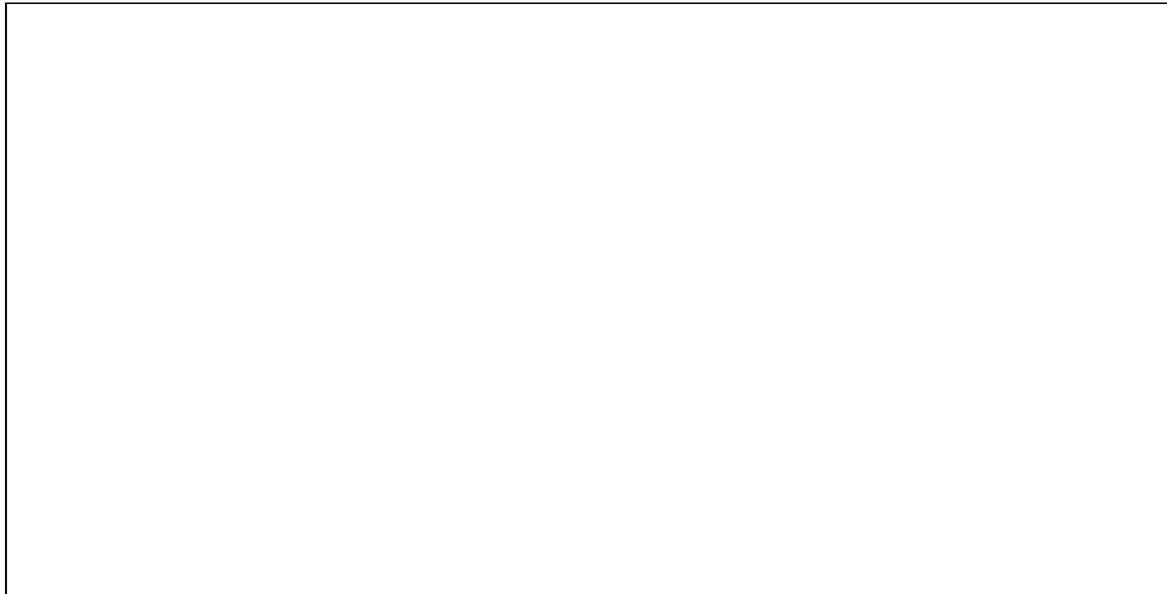


Question 26 Define a `cxr` procedure that takes two arguments: an expression, and a list of `a` and `d` symbols (which stand for `car` and `cdr`). The procedure returns the object referred to by the concatenation of symbols in the list. For example

```
(cxr x '(a d)) == (cadr x)
```

Your solution for `cxr` should **not** be directly recursive (it shouldn't call itself), though it may make use of the recursive procedures `foldl` or `foldr` defined below. Your solution should not use any list procedures except `car, cdr`, and either `foldl` or `foldr`.

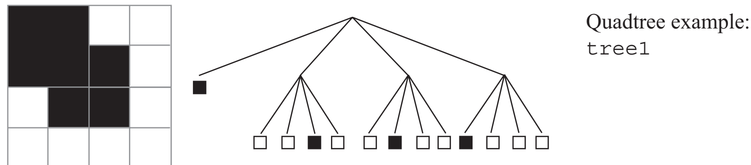
```
(define (foldr f init l)
  (define (helper x l)
    (if (null? l)
        x
        (f (car l) (helper x (cdr l)))))
  (helper init l))

(define (foldl f init l)
  (define (helper x l)
    (if (null? l)
        x
        (helper (f (car l) x) (cdr l))))
  (helper init l))
```



Part 9 (28 points): Data structures:

A **Quadtree** is a popular data structure in computer graphics that allows for the encoding of spatial information using recursive subdivisions of a square. In this problem, we define a quadtree abstract datatype to represent a black-and-white bitmap, where a pixel is either white or black.

The leaves of a quadtree are either black or white, and other nodes have exactly four children representing the four quadrants of the subdivision of the square in the order North-West, North-East, South-West, and South-East. See the example below:



Quadtree example:
tree1

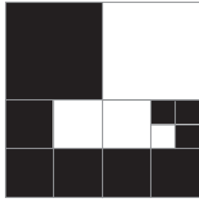
We provide the following abstraction:

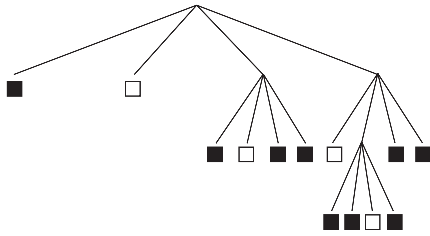
```
(define (black-qt) 'black)
(define (white-qt) 'white)
(define (node-qt nw ne sw se) (list nw ne sw se))

(define (qt-black? qt) (eq? qt 'black))
(define (qt-white? qt) (eq? qt 'white))
(define (qt-node? qt) (list? qt))

(define (qt-nw qt) (car qt))
(define (qt-ne qt) (cadr qt))
(define (qt-sw qt) (caddr qt))
(define (qt-se qt) (cadddr qt))
```

For example, the above tree can be obtained using:

```
(define tree1
  (let ((ne (node-qt (white-qt) (white-qt) (black-qt) (white-qt)))
        (sw (node-qt (white-qt) (black-qt) (white-qt) (white-qt)))
        (se (node-qt (black-qt) (white-qt) (white-qt) (white-qt))))
     (node-qt (black-qt) ne sw se)))
```

Question 27: Write the code to create the following tree :





`tree2`

Depth-first encoding

We want to represent quadtrees in a compact and flat manner using a list composed of 'n' to indicate a node, '0' for black, and '1' for white. More precisely, we define the depth-first encoding of the quadtree as:

- 0 if the tree is a black leaf, and 1 if it is a white leaf
- the concatenation of the symobl `n` and the depth-order encoding of the sub-trees in the order described above (NW, NE, SW, SE) otherwise.

For example, the encoding for tree1 is:

`'(n 0 n 1 1 0 1 n 1 0 1 1 n 0 1 1 1).`

For this, we will write a function `qt-depth-first-encode`, and your job will be to fill in the code for CODE-1, CODE-2, and CODE-3. Be careful to respect the abstraction barrier.

```
(define (qt-depth-first-encode qt)
   (cond ((qt-black? qt) CODE-28)
         CODE-29
         (else CODE-30)))
```

Question 28 Write the code for CODE-28

Question 29: Write the code for CODE-29

Question 30: Write the code for CODE-30

Simplification

Some quadtrees contain redundant information when subdivision was used in regions of constant color. We want to simplify such tree into the most-compact quadtree. We will do this in two steps: first, we simplify nodes that have only leaves as children, next we will recursively simplify the full tree.

First, we want to simplify a node that has four black children or four white children. Write a function `qt-conflate` that takes a node and verifies if its four children are all black (resp. all white), in which case it returns a black (resp. white) leaf. Otherwise, return the node unchanged. You can assume that the argument `n` is not a leaf.

Here are some example usages:

```
(qt-conflate (node-qt (black-qt) (black-qt) (black-qt) (black-qt)))
          -->  black
(qt-conflate (node-qt (white-qt) (black-qt) (black-qt) (black-qt)))
          -->  (white black black black)
```

```
(define (qt-conflate n)
   YOUR-CODE)
```

Question 31: Write the code for YOUR-CODE-HERE to complete the implementation of `qt-conflate`.

Next, we want to recursively simplify the tree using the `qt-conflate` procedure above.

For example

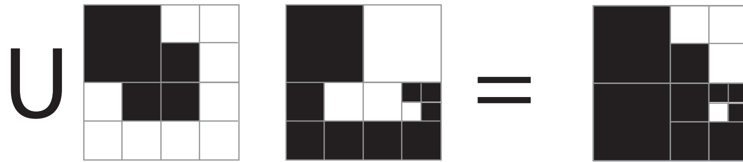
```
  (qt-simplify (node-qt (white-qt)
                        (node-qt (black-qt) (black-qt) (black-qt)
                        (black-qt))
                        (white-qt)
                        (node-qt (white-qt) (black-qt) (black-qt)
                        (black-qt))))
          -->  (white black white (white black black black))
```

```
(define (qt-simplify qt)
   (if (qt-node? qt)
      YOUR-CODE-HERE
      qt))
```

Question 32: Write the code for YOUR-CODE-HERE to complete the `qt-simplify` procedure above.

Union

Finally, we want to take two images described by a quadtree and compute the union of the black parts. Note that the subdivision structure of the union quadtree might be simpler than that of the two sub-trees. In this case, we want the most compact quadtree. For example, the union of tree1 and tree2 is



```
(define (qt-union qt1 qt2)
   (cond ((qt-black? qt1) CODE-33)
         ((qt-white? qt1) CODE-34)
         ((qt-black? qt2) CODE-35)
         ((qt-white? qt2) CODE-36)
         (else  CODE-37)))
```

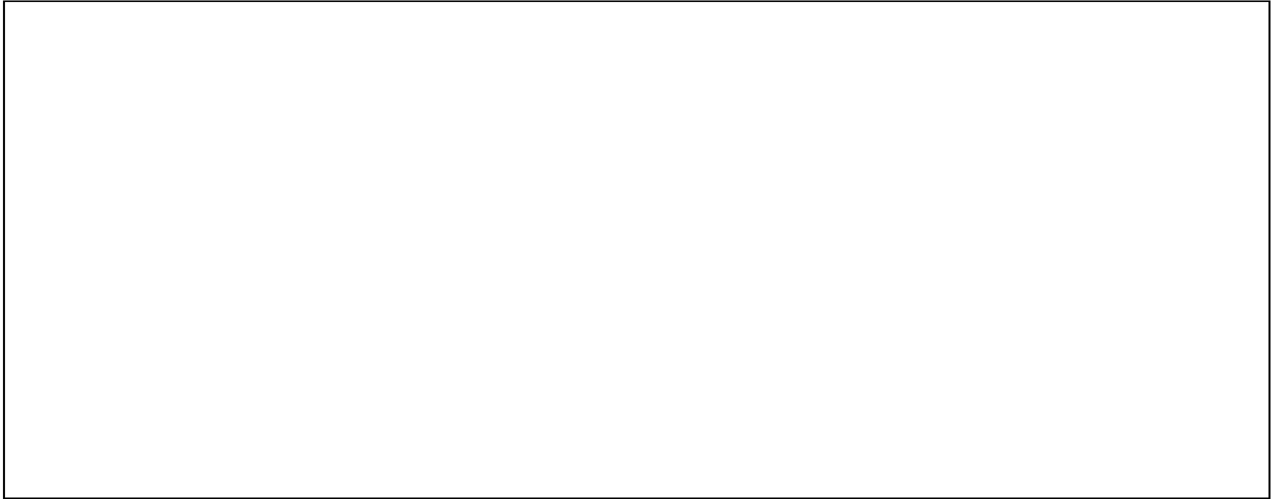
Question 33: Write the code for CODE-33

Question 34: Write the code for CODE-34

Question 35: Write the code for CODE-35

Question 36: Write the code for CODE-36

Question 37: Write the code for CODE-37



Part 10 (10 points): Asynchronous computation

Consider the following procedures and definitions:

```
(define x 2)

(define (proc1) (set! x (+ x 1)))

(define (proc2) (set! x (* x x)))
```

Question 38:

Suppose that `proc1` and `proc2` are being applied asynchronously, with no serialization. What is the set of possible values for `x` at the completion of both evaluations?

Question 39: Suppose that the `set!` expressions in both `proc1` and `proc2` in the previous question are serialized by the same serializer, and that the two procedures are being applied asynchronously. What is the set of possible values for `x` at the completion of both evaluations?

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

0.1 The Core Evaluator

```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)(make-procedure (lambda-parameters exp) (lambda-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (m-eval (cond->if exp) env))
        ((application? exp)
         (m-apply (m-eval (operator exp) env) (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              arguments
                              (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env)))))

(define (eval-if exp env)
  (if (m-eval (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (m-eval (assignment-value exp) env)
                       env))

(define (eval-definition exp env)
```

```
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))
```

0.2 Representing Expressions

```
(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))

(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))

(define (variable? exp) (symbol? exp))
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))   (cadr exp)   (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) (cddr exp))))  ; formal params, body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (cddr lambda-exp))
(define (make-lambda parms body) (cons 'lambda (cons parms body)))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp))) (cadddr exp) 'false))
(define (make-if pred conseq alt) (list 'if pred conseq alt))

(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define (sequence->exp seq)
  (cond ((null? seq) seq)
```

```
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin exp) (cons 'begin exp))

(define (application? exp) (pair? exp))
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
```

0.3 Representing procedures

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? exp)
  (tagged-list? exp 'procedure))
(define (procedure-parameters p) (list-ref p 1))
(define (procedure-body p) (list-ref p 2))
(define (procedure-environment p) (list-ref p 3))
```

0.4 Representing environments

```
;; Implement environments as a list of frames; parent environment is
;; the cdr of the list.  Each frame will be implemented as a list
;; of variables and a list of corresponding values.
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values) (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
```

```
        (error "Unbound variable -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))

(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))        ; Same as lookup except for this
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))


(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding-to-frame! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

0.5 Primitive Procedures and the Initial Environment

```
(define (primitive-procedure? proc) (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list '+ +)
        (list '> >)
        (list '= =)
        (list '* *)
        (list 'display display)
        (list 'not not)
        ; ... more primitives
        ))

(define (primitive-procedure-names) (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc))) primitive-procedures))
(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))
```

```
(define (setup-environment)
  (let ((initial-env (extend-environment (primitive-procedure-names)
                                         (primitive-procedure-objects)
                                         the-empty-environment)))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
(define the-global-environment (setup-environment))
```

0.6 The Read-Eval-Print Loop

```
(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-environment)))
      (announce-output output-prompt)
      (display output)))
  (driver-loop))
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```