Choose the answer which best describes each of the following:

(a) You and a friend decide to have lunch at a rather popular Berkeley resturant. Since there is a long line at the service counter, each group of people entering the restaurant decide to have someone grab a table while someone else waits in the line to order the food. This sounds like a good idea, so your friend sits down at the last free table while you get in line. Unfortunately, the resturant stops taking orders when there are no tables available, and you have to wait in line for the people ahead of you. This is an example of

_______incorrect answer

_______deadlock

_______inefficency (too much serialization)

_______unfairness

_______none of the above (correct parallelism)

(b) After you finally get to eat lunch, you and your friend decide to go to the library to work on a joint paper. The library has a policy that students who enter the library must open their backpacks to show that they are not bringing food into the library. They have several employees doing backpack inspection, so there are several lines for people waiting to be inspected. However, today there was a bomb threat, and so the inspectors also use a handheld metal detector to examine the backpacks. Although there are several inspectors, the library only has one metal detector. This is an example of

_______incorrect answer

_______deadlock

_______inefficency (too much serialization)

_______unfairness

_______none of the above (correct parallelism)

(c)While you are working on the paper, your friend decides to do some research for your paper and leaves for a few hours while you continue writing. This is an example of

_______incorrect answer

_______deadlock

_______inefficency (too much serialization)

_______unfairness

_______none of the above (correct parallelism)

We want this behavior:

```
> (define m1 (make-marble 'red))
> (define m2 (make-marble 'blue))
> (define m3 (make-marble 'yellow))
> (m1)
(yellow blue red)
```

Whenever *any* marble is invoked, it returns a list of *all* the marble colors. Which of the following definitions is correct:

```
____ (define make-marble
       (lambda (color)
         (let ((all-colors '()))
           (lambda ()
             (set! all-colors (cons color all-colors))
             all-colors))))

____ (define make-marble
       (let ((all-colors '()))
         (lambda (color)
           (lambda ()
             (set! all-colors (cons color all-colors))
             all-colors))))

____ (define make-marble
       (lambda (color)
         (let ((all-colors '()))
           (set! all-colors (cons color all-colors))
           (lambda ()
             all-colors))))

____ (define make-marble
       (let ((all-colors '()))
         (lambda (color)
           (set! all-colors (cons color all-colors))
           (lambda ()
             all-colors))))
```

When people want to get married in a hurry, they can go to Las Vegas where marriage licenses are issued quickly. We wish to serialize the marriage ceremony in Las Vegas.

(a) When a man and a woman wish to get married they are each placed in the back of a queue. The man is placed in the `man-queue` and the woman is placed in the `woman-queue`. There they wait until they are dequeued by a Justice of the Peace, who is allowed to perform marriages.

```
;;Using queues from Week 9, page 262 in SICP

(define man-queue (make-queue))
(define woman-queue (make-queue))

(define (vegas-goers man woman)
  (insert-queue! man man-queue)        ;;Put man at back of queue
  (insert-queue! woman woman-queue))   ;;Put woman at back of queue

(define (insert-queue! queue item)     ;;This procedure is from SICP
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```

Is there any need to add serializers in the `vegas-goers` procedure? In other words, is the following implementation dangerous? Check the **best** answer:

```
(parallel-execute (lambda () (vegas-goers 'Paul 'Linda))
                  (lambda () (vegas-goers 'John 'Yoko)))
```

_______No, because men and women are contained in different queues.

_______Yes, because we could get incorrect results.

_______Yes, to avoid deadlock between the two queues for men and women.

_______No, because the insertion algorithm for queues has no critical section.

(b) There are several Justices of the Peace in Las Vegas. Only Justices can perform marriages, so only Justices can access the two queues. To perform the ceremony a Justice must dequeue a man and a woman from their respective queues. Here is the code:

```
(define (justice name)
  (let ((groom (front-queue man-queue))      ;;Groom at from of his queue
        (bride (front-queue woman-queue)) )  ;;Bride at front of her queue
    (delete-queue! man-queue)                ;;Remove from queues
    (delete-queue! woman-queue)
    (marry groom bride))
```

```
(define (front-queue queue)                  ;; Procedures from SICP
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))

(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else (set-front-ptr! queue (cdr (front-ptr queue)))
              queue)))
```

We need to serialize the `justice` procedure. Alyssa P. Hacker suggests we use a single serializer for the entire Justice procedure, as follows:

```
(define s (make-serializer))
(define (justice name)
  ((s (lambda () (let ((groom (front-queue man-queue))
                       (bride (front-queue woman-queue)))
                   (delete-queue! man-queue)
                   (delete-queue! woman-queue)))))
  (marry groom bride)
  (display "Congratulations!"))
```

Here are some arguments regarding Alyssa's strategy. Check the **best** answer:

_______This is inefficient. Since only deleting from a queue actually changes the contents of the queue, we do not have to serialize until we call `delete-queue!`.

_______This is incorrect. Because queues, not justices, are responsible for adding and removing people, all serialization should take place within the `queue` ADT.

_______This can cause deadlock. Because `let` is syntactic sugar for a procedure and procedure invocation, the `let` form in the `justice` procedure is actually two instructions.

_______This is correct.

This question is about an abstract data type for sorted lists. A sorted list is just like a regular list, except that the elements are always kept in sorted order.

(a) We want to be able to use sorted lists this way:

```
> (define slist1 (make-empty-sorted-list))
> (define slist2 (make-empty-sorted-list))

> ; sorted->regular converts a sorted list into a regular Scheme list
> (sorted->regular slist1)
()

> ; insert! inserts its first argument into its second argument, in sorted
> ; order.
> (insert! 1 slist1)
> (insert! 5 slist2)
> (sorted->regular slist1)
(1)
> (sorted->regular slist2)
(5)
> (insert! 3 slist2)
> (insert! 7 slist2)
> (sorted->regular slist2)
(3 5 7)
```

We define this constructor:

```
(define (make-empty-sorted-list)
  '())
```

With this constructor, can we define `insert!` so that it behaves as shown in the example above? Check the **best** answer:

_______Yes, `insert!` could be defined so that everything works in the example above.

_______No, `insert!` can't be defined to work as it does in the example above because there is nothing to mutate.

_______No, `insert!` can't be defined to work as it does in the example above because both sorted lists will share the same memory.

_______No, `insert!` can't be defined to work as it does in the example above because mutation can only put new entries at the beginning of the list.

(b) We want to be able to use sorted lists this way (note that the initial list creation is done differently):

```
> (define slist1 the-empty-sorted-list)
> (define slist2 the-empty-sorted-list)

> (sorted->regular slist1)
()

> (insert! 1 slist1)
> (insert! 5 slist2)
> (sorted->regular slist1)
(1)
> (sorted->regular slist2)
(5)
> (insert! 3 slist2)
> (insert! 7 slist2)
> (sorted->regular slist2)
(3 5 7)
```

We use this definition:

```
(define the-empty-sorted-list
  (cons 'sorted-list '()))
```

With this definition, can we define `insert!` so that it behaves as shown in the example above? Check the **best** answer:

_______Yes, `insert!` could be defined so that everything works in the example above.

_______No, `insert!` can't be defined to work as it does in the example above because there is nothing to mutate.

_______No, `insert!` can't be defined to work as it does in the example above because both sorted lists will share the same memory.

_______No, `insert!` can't be defined to work as it does in the example above because mutation can only put new entries at the beginning of the list.

Suppose that `my-tree` is the following tree:

```
           1
          /|\
         / | \
        2  5  3
        |    / \
        4   6   7
```

You are given the following procedures:

```
(define (foo tree)
  (if (null? (children tree))
      (datum tree)
      (foo (car (children tree)))))

(define (baz tree)
  (map datum (children tree)))

(define (garply tree)
  (make-tree 1 (map garply (children tree))))
```

What is the value returned by each of the following? If it's a tree, draw a picture of it, as above; if not, show how Scheme will print the value.

(a) `(foo my-tree)`

(b) `(baz my-tree)`

(c) `(garply my-tree)`

```
(define-class (scoop flavor)
  ; maybe (parent (cone)) -- see part (A) below
  )

(define-class (vanilla)
  (parent (scoop 'vanilla)))
(define-class (chocolate)
  (parent (scoop 'chocolate))

(define-class (cone)
  ; maybe (parent (scoop)) -- see part (A) below
  (instance-vars (scoops '()))
  (method (add-scoop new)
    (set! scoops (cons new scoops)))
  (method (flavors)
    (map ____see (B) below____ scoops)))
```

(A) Which of the `parent` clauses shown above should be used?

_______The `scoop` class should have `(parent (cone))`.

_______The `cone` class should have `(parent (scoop))`.

_______Both.

_______Neither.

(B) What is the missing expression in the `flavors` method?

(C) Which of the following is the correct way to add a scoop of vanilla ice cream to a cone named `my-cone`?

_______`(ask my-cone 'add-scoop 'vanilla)`

_______`(ask my-cone 'add-scoop vanilla)`

_______`(ask my-cone 'add-scoop (instantiate 'vanilla))`

_______`(ask my-cone 'add-scoop (instantiate vanilla))`

Suppose we want to write a procedure `prev` that takes as its argument a procedure `proc` of one argument. `Prev` returns a new procedure that returns the value returned by *the previous call to* `proc`. The new procedure should return `#f` the first time it is called. For example:

```
> (define slow-square (prev square))
> (slow-square 3)
#f
> (slow-square 4)
9
> (slow-square 5)
16
```

Which of the following definitions implements `prev` correctly? **Pick only one.**

```
______ (define (prev proc)
         (let ((old-result #f))
           (lambda (x)
             (let ((return-value old-result))
               (set! old-result (proc x))
               return-value))))


______ (define prev
         (let ((old-result #f))
           (lambda (proc)
             (lambda (x)
               (let ((return-value old-result))
                 (set! old-result (proc x))
                 return-value)))))


______ (define (prev proc)
         (lambda (x)
           (let ((old-result #f))
             (let ((return-value old-result))
               (set! old-result (proc x))
               return-value))))


______ (define (prev)
         (let ((old-result #f))
           (lambda (proc)
             (lambda (x)
               (let ((return-value old-result))
                 (set! old-result (proc x))
                 return-value)))))
```

Question 4 (8 points):

Consider the following computation:

```
> (define x 5)

> (define (baz z)
    (let ((x 100)
          (function (lambda (y) (+ x y))))
      (function (* x z))))
```

```
> (baz 7)
```

(a) Here is an environment diagram representing the result of this computation, but with a few missing pieces. Complete the diagram by showing the following:

- The values of `y` and `function`.
- Arrows from the right bubbles of two procedures.
- Arrows from two frames showing what environments they extend.

(b) What is the value returned by `(baz 7)`? ______________________________

Write the procedure `same-parity!` that takes as its argument a list of integers, and modifies the list by mutation so that it contains only those elements of the same parity (even or odd) as the first element:

```
> (same-parity! (list 3 8 1 5 2 9 4 6 7 1))
(3 1 5 9 7 1)
```

Do not allocate any new pairs! Remove elements of the wrong parity from the argument list by mutation. Don't forget to return the resulting list as the return value from your procedure.

You may use this helper procedure:

```
(define (same x y)
  (= (remainder x 2) (remainder y 2)))
```

What will the Scheme interpreter print in response to **the last expression** in each of the following sequences of expressions? Also, draw a "box and pointer" diagram for the result of each printed expression. If any expression results in an error, **circle the expression that gives the error message.** Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(define f (list 2 3))
(define g (append f f))
(set-car! g f)
g
```

```
(let ((x (list 1 2 3)))
   (set-car! x (list 'a 'b 'c))
   (set-car! (cdar x) 'd)
   x)
```

```
(define x 3)
(define m (list x 4 5))
(set! x 6)
m
```

(a) Here are some situations that might be simulated using OOP. In each case we want to know whether class A should be a parent of class B (answer "Yes" or "No"):

• We're simulating a kitchen. Class A: silverware. Class B: fork.

• We're simulating a shopping mall. Class A: food court. Class B: restaurant.

• We're simulating a library. Class A: bookshelf. Class B: book.

(b) In each of the following situations, should the given variable be a class variable or an instance variable (answer "Class" or "Instance")?

• In the shoe class, the total number of shoes in the world.

• In the refrigerator class, the maximum safe temperature.

• In the person class in the adventure game, the person's favorite color.

What are the possible values of variable a after each of the following parallel executions? **If deadlock is a possibility, say so.**

(a)

```
(define a 2)
(parallel-execute
   (lambda () (set! a (list a a)))
   (lambda () (set! a (list a 1))))
```

(b)

```
(define a 2)
(define s (make-serializer))
(parallel-execute
   (s (lambda () (set! a (list a a))))
   (s (lambda () (set! a (list a 1)))))
```

(c)

```
(define a 2)
(define s (make-serializer))
(define t (make-serializer))
(parallel-execute
   (s (lambda () (set! a (list a a))))
   (t (lambda () (set! a (list a 1)))))
```

Suppose there are N students taking a midterm. Suppose we have a vector of size N, and each element of the vector represents one student's score on the midterm. Write a procedure `(histogram scores)` that takes this vector of midterm scores and computes a histogram vector. That is, the resulting vector should be of size M+1, where M is the maximum score on the midterm (it's M+1 because scores of zero are possible), and element number I of the resulting vector is the number of students who got score I on the midterm.

For example:

```
> (histogram (vector 3 2 2 3 2))
#(0 0 3 2) ;; no students got 0 points, no students got 1 point,
           ;; 3 students got 2 points, and 2 students got 3 points.
> (histogram (vector 0 1 0 2))
#(2 1 1) ;; 2 students got 0 points, 1 student got 1 point,
         ;; and 1 student got 2 points.
```

Do not use `list->vector` or `vector->list`.

Note: You may assume that you have a procedure `vector-max` that takes a vector of numbers as argument, and returns the largest number in the vector.

Question 6 (7 points): For reference, here is the insertion sort program you saw in week 3, rewritten to use lists instead of sentences:

```
(define (sort lst)
  (if (null? lst)
      '()
      (insert (car lst) (sort (cdr lst)))))

(define (insert value sorted)
  (cond ((null? sorted) (list value))
        ((< value (car sorted)) (cons value sorted))
        (else (cons (car sorted) (insert value (cdr sorted))))))
```

We are going to rewrite this to use list mutation, rearranging the order of the pairs in the spine of the argument list; **the procedure will return the sorted list.** Don't worry about preserving the value of any variables that point to the original list. Sample use:

```
> (sort! (list 7 3 87 5))
(3 5 7 87)
```

Your job is to fill in the blanks in the partial definition below. You probably will not need all the space given. **Note that the first argument to `insert!` is the pair whose car is the value to be inserted, not the value itself.**

Your procedure must not create new pairs!

```
(define (sort! lst)
  (if (null? lst)
      '()
      (insert! lst (sort! (cdr lst)))))

(define (insert! value-pair sorted)
  (cond ((null? sorted)
         (set-cdr! value-pair '())
         value-pair)
        ((< (car value-pair) (car sorted))

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________)

        (else

         ______________________________________________________

         ______________________________________________________

         ______________________________________________________)))
```

(a) Draw the environment diagram showing the situation after both of the following expressions have been evaluated:

```
(define x 17)

(define f
  (let ((x 4))
    (lambda (y)
      (print x)
      (set! x y)
```

```
     y)))
```

(b) Show how the environment is changed when the following expression is evaluated. Also, what is printed out and what is returned?

```
(f 5)
```

(c) Suppose we evaluated `(f 5)` in a Scheme that had dynamic scoping. What is printed out and what is returned?

Consider a list of numbers, such as `(9 17 25 3 21)`. We would like to sort this list into ascending numerical order, *in place*—that is, using the existing list structure, without doing new `cons`es. Here is the algorithm: If the list is empty, it's sorted. Otherwise, exchange the `car` of the list with the smallest number in the list. Then recursively sort the `cdr` of the list. For instance, if we started with

then the first step of the recursion will do this:

Here is part of the program:

```
(define (sort! nums)
  (if (null? nums)
      nil
      (sequence
       (exch! nums (min-tail nums))
       (sort! (cdr nums)))))
```

`Min-tail` finds the smallest number in the list. Its returned value is not the number itself, but the sublist whose head is that number.

```
(define (min-tail nums)
  (define (iter min-so-far min-sublist rest)
    (cond ((null? rest) min-sublist)
          ((< (car rest) min-so-far) (iter (car rest) rest (cdr rest)))
          (else (iter min-so-far min-sublist (cdr rest)))))
  (iter (car nums) nums (cdr nums)))
```

Your job is to write `exch!`, the procedure that takes the head of a list and a sublist of the list as arguments, and exchanges the two numbers at the heads of those lists.

You are to write a procedure `cross` that takes three arguments: the first is a function of two arguments, and the other two are lists. The job of `cross` is to apply the function argument to every possible pair of values, choosing the first value from the first list and the second value from the second list. For example:

```
==> (cross + '(1 2 3) '(40 50))
(41 42 43 51 52 53)
```

The length of the returned list will be the product of the lengths of the two argument lists. Don't worry about the order of elements in the returned list; if your `cross` returns

```
(41 51 42 52 43 53)
```

instead, that's okay.

After writing `cross`, use it to generate a list of all 52 cards in a deck, with each card represented as a pair:

```
((ace . hearts) (2 . hearts) (3 . hearts) ... (jack . diamonds) ...)
```

Remember the problem about moving from room to room in an adventure game on the second midterm? In that problem we asked you to use streams to model the passage of time, with a stream of directions as argument and a stream of rooms visited as the returned value.

This time you are going to model the same situation using local state. As before, you are given a function `next-room` that takes as arguments a room number and a direction; its result is the room where you end up if you move in the given direction from the given room:

```
==> (next-room 14 'South)
9
```

means that room 9 is south of room 14. You are to write a procedure `make-player` that creates a player object. This object (i.e., a procedure) should accept messages like `South` and should move from its current position in the indicated direction. It should remember the new location as local state, and should also return the new location as its result. The argument to `make-player` is the initial room:

```
==> (define Frodo (make-player 14))
FRODO
==> (Frodo 'South)
9
==> (Frodo 'South)
26
```

You *must* make each move relative to the result of the previous move, not starting from the initial room each time!

Draw a "box and pointer" diagram for the following Scheme expressions:

a (1 point).

```
(let ((x (cons nil nil)))
  (set-car! x x)
  (set-cdr! x x)
  x)
```

b (2 points).

```
(define (funny! x)
  (if (null? x)
      nil
      (let ((temp (cdr x)))
        (funny! temp)
        (set-cdr! x (car x))
        (set-car! x temp)
        x))
  (funny '(1 2)))
```

c (2 points). Write a Scheme expression to construct the following structure.

```
           .-----------------------------.
           |                             |
           V                             |
         ---------         ---------     |
         |   |   |         |   |  /|     |
-------->| | | ----------->| | | / |     |
         | | |   |         | | |/  |     |
         --|------         --|------     |
           |   ^             |           |
           |   |             |           |
           |   |             |           |
           V   |             V           |
         ------|--         ---------     |
         |   | | |         |   |   |     |
         | | | | |         | | | --------'
         | | |   |         | | |   |
         --|------         --|------
           |                 |
           |                 |
           |                 |
           V                 V

           A                 B
```

For a statistical project you need to compute lots of random numbers in various ranges. (Recall that `(random 10)` returns a random number between 0 and 9.) Also, you need to keep track of *how many* random numbers are computed in each range. You decide to use object-oriented programming; the procedure `make-random-generator` will return an object that accepts two messages. The message `number` means "give me a random number in your range" while `count` means "how many `number` requests have you had?" The numeric argument to `make-random-generator` specifies the range of random numbers for this object, so

```
(define r10 (make-random-generator 10))
```

will create an object such that `(r10 'number)` will return a random number between 0 and 9, while `(r10 'count)` will return the number of random numbers `r10` has created.

```
(define (make-random-generator n)
  (let ((number-of-calls 0))
    (define (number)
      (set! number-of-calls (1+ number-of-calls))
      (random n))
    (define (count)
      number-of-calls)
    (define (dispatch op)
      (cond ((eq? op 'number) (number))
            ((eq? op 'count) (count))
            (else (error "Unknown operator."))))
    dispatch))
```

a (3 points). Draw an environment diagram after evaluating the call

```
(define rng (make-random-generator 5))
```

b (3 points). Draw an environment diagram, and indicate which is the current environment, while in the `count` procedure during the evaluation of

```
(rng 'count)
```

c (1 point). How many frames are created each time you evaluate

```
(rng 'number)
```

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression:

```
(let ((x (list 1 2 3)))
  (set-car! (cdr x) 4)
  x)
```

```
(let ((x (list 1 2 3)))
  (set-cdr! (car x) 4)
  x)
```

```
(let ((x (list 1 2 3)))
  (set-cdr! x 4)
  x)
```

```
(let ((x (list 1 2 3)))
  (set-car! (cdr x) x)
  x)
```

Draw an environment diagram showing the result of defining the following procedure. (The purpose of the procedure is that each time it's called, it returns `#t` if it has already been called with the same argument value.)

```
(define duplicate?
  (let ((values '()))
    (lambda (test)
      (cond ((memq test values) #t)
            (else (set! values (cons test values))
                  #f) ))))
```

We are going to invent a simplified adventure game. In this version, there are no things. People are represented as objects; places are just symbols. All you can do is move from one place to another; there are no things and no interactions with other people.

The connections between places are represented in a table, as in data-directed programming. That is, to indicate that you can get from Evans to PSL by going east, we'll say

```
(put 'Evans 'east 'PSL)
```

Your job is to define the `person` class, that takes an initial place as its argument. The resulting person object accepts messages like `east` and returns the new location of the person:

```
> (define Brian (instantiate person 'BH-Office))
brian
> (ask Brian 'down)
60a-lab
> (ask Brian 'east)        ;; a request for which there is no PUT defined
cant-get-there-from-here
> (ask Brian 'down)
Evans
```

You do *not* have to write `put`, `get`, `ask`, or anything else that you've seen in the book or handouts. Assume that all the needed connections between rooms have already been established with `put`.

---

a) (2 pts) What will Scheme print in response to the following expressions? Assume that they are typed in sequence (we've done the first two for you!). Although they will not be graded, you will find it helpful to draw the corresponding box and pointer diagrams. If an expression produces an error message, you may just say "error"; you don't have to provide the exact text of the message.

```
> (define x (cons 1 nil))
x
```

```
> (define y (cons 2 3))
y
> (sequence (set-cdr! x y) x)

> (sequence (set-car! y (car x)) x)

> (sequence (set-cdr! (cdr x) (cdr x)) y)

> (sequence (set-car! (car x) 5) (car x))
```

b) (3 pts) For this part, you are to criticize the implementation of the function `new-set!`, which is supposed to behave *exactly* like `set!`. (Hint: there are two major screwups.)

```
(define (new-set! x y)
   (if (and (pair? x) (pair? y))
      (sequence
         (set-car! x (car y))
         (set-cdr! x (cdr y)))
      (set! x y)))
```

Illustrate your answer by giving expressions or sequences of expressions using `new-set!` that result in output different from what would be obtained using `set!`.

---

a) (2 pts) Implement the function `(make-mod-counter n)` that returns a particular kind of counter procedure. The procedure returned should start with an internal counter with a value of 0, and deal with the following messages:
`up` increases the counter by 1, modulo `n`
`down` decreases the counter by 1, modulo `n`
`reset` sets the counter back to 0.
Each time the counter procedure is called, it should return the new value of the internal counter. Here is a small sample of the desired bahaviour:

```
> (define binary-counter (make-mod-counter 2))
binary-counter
> (binary-counter 'up)
1
> (binary-counter 'up)
0                          ;;; since 2 mod 2 is 0
```

(You may use the primitive function `modulo`: eg `(modulo 27 5) = 2`).)

b) (3 pts) Now complete the following to show the final environment diagram resulting from the following sequence of commands (use the evaluation rules summarised on p.188 of Abelson and Sussman):

```
> (define tc (make-mod-counter 3))
tc
> (tc 'up)
1
```

Write a procedure `circulate!` that takes a list (perhaps a list of lists to any depth) as argument. It should mutate the list (that is, change the contents of the existing pairs without constructing any new pairs) so that the list and all of its sublists become circular, with the last element repeated infinitely. For example,

```
(circulate! '((a b) (c (d e)) f))
```

should return a list effectively containing

```
((a b b b ...) (c (d e e e ...) (d e e e ...) (d e e e ...) ...)) f f f ...)
```

(Of course a circular list can't really be printed.) In this example, the entire list was made circular by including the last element (`f`) repeatedly; the sublist `(a b)` was circulated by repeating the `b`; the sublist `(c (d e))` was circulated by repeating the `(d e)`; and the `(d e)` was circulated by repeating the `e`.

You may assume that the argument list does not already contain circularities. You may also assume that it is a well-formed (proper) list.

We are going to simulate a Compact Disc (CD) player using object-oriented programming. Use the OOP notation as described in the course reader.

(a) Define a CD object class. Every CD contains the following information: a number identifying the recording, and a list of numbers, one per song, indicating where on the CD that song begins. (For our purposes we can think of positions on the disc in terms of the number of seconds of music that come before it.) To instantiate a CD we'll provide two arguments, the ID number and the timing list:

```
(define With-the-Beatles (instantiate cd 102574 '(0 102 293 542 ...)))
```

The timing list contains one extra number at the end, which is the position at which an additional song would begin if there were one. In other words, this extra number indicates the total time of the CD.

A CD object accepts three messages: `id` asks for the ID number; `songs` asks for the number of songs recorded on the disc; and `index` takes a number as its argument and returns the corresponding number from the timing list. (If the argument is zero it returns the first element of the list, and so on.)

```
> (ask With-the-Beatles 'index 2)
293
```

(b) Now define a CD-player class. A good simulation would be very complicated, mainly because once a CD is playing, the object continues to do work even if it gets no more messages. But we'll only simulate a couple of features. In particular, we won't actually play any songs!

The `load` message takes a CD object as its argument and "loads" that CD into the player.

The returned value is the ID number of the CD. The effect of loading a CD is that later messages to the player refer implicitly to the loaded CD.

The `length` message takes a song number as its argument, and returns the length of that song (the difference between its position and the one after it).

The `goto` message takes a song number as its argument, and returns the position at which that song begins. (A more realistic simulation would actually move the laser beam to that position and begin playing, but we'll just return the position.)

```
> (define my-player (instantiate CD-player))
> (ask my-player 'load With-the-Beatles)
102574
> (ask my-player 'goto 3)
542
> (ask my-player 'length 1)
191                             ;; this is 293 − 102
```

On the next page are five environment diagrams. The very large letter within each global frame is the name of that diagram. For each of the following three instruction sequences, identify the environment diagram that results after the completion of the sequence. Note: a single diagram might match more than one instruction sequence. You may find it helpful to draw your own diagrams on scrap paper. You may tear off the next page if that'll make it easier for you; there is nothing on that page that you need to turn in, unless you've written on the back of it.

1. ________

```
(define (thing x)
  (let ((a 5) (b 6))
    (lambda ()
      (* a (+ b x)) )))
(define fred (thing 3))
(fred)
```

2. ________

```
(define (thing x)
  (let ((a 5))
    (lambda (b)
      (* a (+ b x)) )))
(define fred (thing 3))
(fred 6)
```

3. ________

```
(define (thing x)
  (lambda (a)
    (let ((b 6))
      (* a (+ b x)) )))
(define fred (thing 3))
```

```
(fred 5)
```

(diagrams go here)

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-car! (cdr x) (cddr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-cdr! (cdr x) (caddr x))
  x)
(let ((x (list 1 '(2 3) 4)))
  (set-car! x (caddr x))
  x)
```

Write `make-alist!`, a procedure that takes as its argument a list of alternating keys and values, like this:

```
(color orange zip 94720 name wakko)
```

and changes it, by mutation, into an association list, like this:

```
((color . orange) (zip . 94720) (name . wakko))
```

You may assume that the argument list has an even number of elements. The result of your procedure requires exactly as many pairs as the argument, so you will work by rearranging the pairs in the argument itself. **Do not allocate any new pairs in your solution!**

We are going to modify the adventure game project by inventing a new kind of place, called a *hyperspace*. Hyperspaces are just like other places, connected to neighboring non-hyperspace places in specific directions, except that they behave strangely when someone enters one: The person who entered is sometimes magically transported to another hyperspace. (The hyperspaces must all know about each other, but they are not connected to each other through exits. Each hyperspace is connected to specific neighbors, just as any place is.)

Your job is to define the `hyperspace` class. The class must be defined in a way that allows you to know all of its instances. When a person enters a hyperspace, half the time nothing special should happen, but half the time the hyperspace should ask the person to `go-directly-to` some randomly chosen other hyperspace.

You may use the following auxiliary procedures if you wish:

```
(define (coin-heads?) (= (random 2) 1))

(define (choose-randomly stuff)
  (nth (random (length stuff)) stuff))
```

Do not modify any existing class definitions.

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-car! (cdr x) (cddr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-cdr! (cddr x) (cadr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cddr x) (cons (cadr x) (cdddr x)))
  x)
```

Write `merge!`, a procedure that takes two arguments, each of which is a list of numbers in increasing order. It returns a combined, ordered list of all the numbers:

```
> (merge! (list 3 5 22 26) (list 2 7 10 30))
(2 3 5 7 10 22 26 30)
```

Your procedure must do its work by mutation, changing the pointers between pairs to create the new combined list. The original lists will no longer exist after your procedure is finished.

Note: **Do not allocate any new pairs** in your solution. Rearrange the existing pairs.

In an Adventure game, there are often magical things which have some special effect when a person takes them. For example, the magic wand moves you to the cavern, or the gold ring increases your strength.

Define a `magic-thing` class that's just like the `thing` class except that it takes an extra instantiation variable, a procedure of one argument. When a person `take`s the thing, that procedure should be invoked with the person as its argument.

For example:

```
(define magic-wand
  (instantiate magic-thing
               'wand
               (lambda (person) (ask person 'go-directly-to cavern))))
```

(a) Suppose we say

```
> (define baz 'hi)
> (define s (make-serializer))

> (parallel-execute (s (lambda () (set! baz (word baz baz))))
                    (s (lambda () (set! baz 'bye))))
```

What are the two possible values of `baz` after this finishes?

(b) Now suppose that we change the example to leave out one invocation of the serializer, as follows:

```
> (define baz 'hi)
> (define s (make-serializer))

> (parallel-execute (s (lambda () (set! baz (word baz baz))))
                    (lambda () (set! baz 'bye)))
```

What are *all* of the possible values of `baz` this time?

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-cdr! x (cdr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cddr x) (cdr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-cdr! (cdr x) (cadr x))
  x)
```

An *improper list* is a data structure made of pairs, in which the `cdr` of some pair is anything other than a pair or the empty list.

Write the procedure `deep-fix!` that takes a pair as its argument, examines the data structure headed by that pair, and replaces any improper `cdr` with the empty list. For example:

```
> (deep-fix! '(1 2 (3 4 . 5) (6 (7 . 8) 9) 10 . 11))
(1 2 (3 4) (6 (7) 9) 10)
```

(In the example I've broken the rule about not mutating quoted data, in order to make the example easier for you to read.)

Note: **Do not allocate any new pairs** in your solution. Modify the existing pairs.

We're going to add to the adventure game a new kind of person, called a *wizard.* Wizards can move around in the same way that other people do, but they also remember every place they've ever seen. Once a wizard has been someplace, he can return to that place directly, by magic.

The `wizard` class will accept a `revisit` message, whose argument is *the name of* a place where the wizard has already been. If the argument is valid, the wizard will go directly to the place with that name. If not, print an error message.

Implement the `wizard` class in OOP notation.

Here's an abbreviated definition of the `person` class, to jog your memory:

```
(define-class (person name place)
  (method (person?) #t)
  (method (look-around) ...)
  (method (exits) (ask place 'exits))
  (method (go direction) ...)
  (method (go-directly-to new-place) ...)
  ...)
```

(a) Suppose we say

```
> (define baz 10)
> (define s (make-serializer))

> (parallel-execute (s (lambda () (set! baz (/ baz 2))))
                    (s (lambda () (set! baz (+ baz baz)))))
```

What are the possible values of `baz` after this finishes?

(b) Now suppose that we change the example to leave out the serializer, as follows:

```
> (define baz 10)

> (parallel-execute (lambda () (set! baz (/ baz 2)))
                    (lambda () (set! baz (+ baz baz))))
```

What are *all* of the possible values of `baz` this time?

What will the Scheme interpreter print in response to each of the following expressions?

Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-car! x (cdr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cdr x) (cadr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-cdr! (cddr x) (cdr x))
  x)
```

Question 2 (3 points):

(a) Suppose we say

```
> (define baz 10)
> (define s (make-serializer))

> (parallel-execute (s (lambda () (set! baz 7)))
                    (s (lambda () (set! baz (+ baz baz)))))
```

What are the possible values of `baz` after this finishes?

(b) Now suppose that we change the example to use a separate serializer for each process, as follows:

```
> (define baz 10)
> (define s (make-serializer))
> (define t (make-serializer))

> (parallel-execute (s (lambda () (set! baz 7)))
                    (t (lambda () (set! baz (+ baz baz)))))
```

What are the possible values of `baz` this time?

We are going to prepare a simulation of an FM car radio. To simplify the problem we'll restrict our attention to tuning, not to volume or balance or anything else a radio does. This radio features digital tuning. There are six buttons that can be preset to particular stations; for manual tuning, there are `up` and `down` buttons that move to the next higher or lower frequency. (FM frequencies are measured in megahertz and have values separated by 0.2: 88.1, 88.3, 88.5, 88.7, 88.9, 90.1, etc.) To simplify the problem further, we'll ignore the boundary problem of what to do when you're at the lowest assigned FM frequency and try to go `down` below that frequency. Just pretend you can keep going up or down forever.

Use the OOP language (`define-class` and so on).

(a) Create a `button` object class that accepts these messages:

`set-freq! 93.3`	sets the button's remembered frequency

`freq`	returns the remembered frequency

The initial frequency should be zero (because the buttons don't have settings initially).

(b) Create a `radio` object class that has six buttons, numbered 0 through 5, and accepts these messages:

`set-button! 3`	sets button 3 to the radio's current frequency
`push 3`	sets the radio to button 3's frequency
`up`	sets the radio to the next higher frequency
`down`	sets the radio to the next lower frequency
`freq`	returns the radio's current frequency

The radio's initial frequency should be 90.7 MHz. Points to remember: Your radio has to use six of your button objects; you needn't check for invalid argument values in the methods. **Hint:** Give your radio a list of six buttons, and use `list-ref` to get the one you want.

You are given a binary tree, in which the nodes are represented in the form indicated by these selectors:

```
(define datum car)
(define left-branch cadr)
(define right-branch cddr)
```

The empty tree is represented by the empty list.

Write `traverse!`, a procedure that takes a binary tree as its argument, and rearranges the pairs to form an inorder traversal — a linear sequence of the data from the tree, in left-to-right order. (If the tree is a binary search tree, for example, then the result will be a sorted sequence.)

A binary tree with N nodes contains $2N$ pairs. You will string together the pairs containing the data, and discard the pairs containing the branch pointers (after you've collected the data from those branches).

Note: **Do not allocate any new pairs** in your solution. Modify the existing pairs.

Note: In this problem you are changing a structure's abstract data type, from tree to sequence. In such situations, data abstraction doesn't make much sense; just use `car`, `set-cdr!`, etc.

The following expressions were entered into the Scheme interpreter:

```
> (define x (list 'to 'be))
x
> (define y (list 'or 'not))
y
> (define z (cons 1 2))
```

```
z
```

Fill in the blanks with pair mutation requests (one expression per line) so that the final result is as shown. **(Do not construct any new pairs!)**

```
> ______________________________________________

> ______________________________________________

> ______________________________________________
> z
```
~~((to be) or not to be)~~

Lem E. Tweakit writes the following procedure to print structures built out of pairs, using dot notation instead of the list shorthand, so, for example, instead of `(a b c)` it prints `(a . (b . (c . ())))`:

```
(define (print-pair p)
  (if (not (pair? p))
      (display p)
      (begin
        (display "(")
        (print-pair (car p))
        (display " . ")
        (print-pair (cdr p))
        (display ")"))))
```

Ben Bitdiddle tells Lem that his procedure won't work for a circular structure.

(a) Write one or more Scheme expression(s) that will produce *a single pair* that Lem's `print-pair` can't print; draw the box and pointer diagram for that pair; and show the result of calling `print-pair` for that pair.

(b) Lem tries to fix Ben's objection by rewriting his procedure this way:

```
(define (print-pair p)
  (let ((looked-at '()))
    (cond ((not (pair? p)) (display p))
          ((memq p looked-at) (display "#<repeated>"))
          (else
            (set! looked-at (cons p looked-at))  ; put p into looked-at
            (display "(")
            (print-pair (car p))
            (display " . ")
            (print-pair (cdr p))
            (display ")")))))
```

Ben says, "That's the general idea, but your procedure still won't quite work."

Fix Lem's procedure, changing it as little as possible. Then show how your modified procedure will print the pair you created in part (a).

Here is a proposed solution to the dining philosophers problem, supposing there are five philosophers and five chopsticks. Chopstick 0 is to the left of philosopher 0, etc.

```
(define chopstick-serializers (list (make-serializer)
                                    (make-serializer)
                                    (make-serializer)
                                    (make-serializer)
                                    (make-serializer))

(define chopsticks (list #f #f #f #f #f))   ; will be #T if chopstick busy

(define (philosopher num)
  (think)
  (get-chopstick num)                       ; left chopstick
  (get-chopstick (remainder (+ num 1) 5))   ; right chopstick
  (eat)
  (release-chopstick num)
  (release-chopstick (remainder (+ num 1) 5))
  (philosopher num))

(define (get-chopstick num)
  (if (((list-ref chopstick-serializers num)   ; find the serializer
        (lambda ()                             ; serialize this procedure
          (let ((ch ((repeated cdr num) chopsticks)))
            (if (car ch)                       ; if chopstick is in use...
                #t                             ;  return #T to the IF
                (begin
                 (set-car! ch #t)              ; otherwise mark it busy
                 #f))))))                      ;  and return #F to the IF
      (get-chopstick num)))

(define (release-chopstick num)
  (set-car! ((repeated cdr num) chopsticks) #f))

(parallel-execute (lambda () (philosopher 0))
                  (lambda () (philosopher 1))
                  (lambda () (philosopher 2))
                  (lambda () (philosopher 3))
                  (lambda () (philosopher 4))
```

(a) Suppose that `get-chopstick` is correct (and note that it provides serialization). Does `philosopher` exhibit (pick the correct answer):

(A) possible incorrect results

(B) possible deadlock

(C) inefficiency (reduced parallelism)

(D) none of the above

(b) Suppose that `philosopher` is correct. Does `get-chopstick` exhibit (pick the correct answer):

(A) possible incorrect results

(B) possible deadlock

(C) inefficiency (reduced parallelism)

(D) none of the above

In lab you saw how `make-cycle` (*SICP*, page 256) makes a list into a cycle:

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)

(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))
```

Now, we ask you to undo that process. Define a function `unmake-cycle` that takes in a non-empty cycle `L` (made by `make-cycle`) and breaks the cycle. The pair `L` should be at the head of the list that `unmake-cycle` returns. **Do not allocate any new pairs!**

```
> (define school (list 'u 'c 'berkeley))
> school
(u c berkeley)
> (make-cycle school)   ;; return value omitted; we only want side-effect.
                        ;; school is now (u c berkeley u c berkeley ...)
> (unmake-cycle school)
(u c berkeley)
> school
(u c berkeley)
```

We are going to show you an interaction with the Scheme interpreter. Your job is to infer the structure of the two lists `U` and `C` and draw their box and pointer representation.

```
> U
(cal (golden) bears)
> C
((golden) bears)

> (eq? (cdr U) C)
```

```
#f
> (eq? (cadr U) (car C))
#t
> (eq? (cddr U) (cdr C))
#t
```

(a) Draw the box-and-pointer representation of `U` and `C` in the space below.

U

C

(b) **Without changing your answer to (a) above, copy it below.** Then suppose we evaluate the following expressions:

```
> (set-car! (cdr C) (car U))
> (set-car! (cdr U) 'beloved)
```

Draw the final box-and-pointer representation of `U` and `C` in the space below.

U

C

(c) Fill in the blanks to indicate the final printed values of `U` and `C`.

```
> U
```

____

```
> C
```

____

The textbook provides the following definition of memoization:

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previous-result (lookup x table)))
        (or previous-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

...but Louis Reasoner has lost his book! He tries to define `memoize` as the following:

```
(define memoize                 ;; this line changed (no parens or f)
  (let ((table (make-table)))
    (lambda (f)                  ;; this line added
      (lambda (x)
        (let ((previous-result (lookup x table)))
          (or previous-result
              (let ((result (f x)))
                (insert! x result table)
                result)))))))
```

Louis takes the usual `fib` procedure:

```
(define (fib x)
  (if (< x 2)
      x
      (+ (fib (- x 1))
         (fib (- x 2)))))
```

He then tests his version of `memoize` by memoizing `fib` exactly as in the book:

```
(define memo-fib
   (memoize (lambda (x)
              (if (< x 2)
                  x
                  (+ (memo-fib (- x 1))
                     (memo-fib (- x 2)))))))
```

Louis tries out his code, and traces through it. To his surprise, it seems to work! His `memo-fib` computes the answer in $O(n)$ time! Ben Bitdiddle looks at his code and comments: "Louis, your code has a major flaw in it..."

Does Louis' `memoize` give wrong answers?

____Yes ______________________________

If yes, explain why, and give an example using Louis' `memoize` that will return an incorrect result.

If no, explain what flaw Ben means, and give an appropriate example.

For full credit, your explanation must be no more than 20 words. If you give two explanations, we will grade the less correct one.

We have defined these three numeric variables and four procedures:

```
(define r  10)
(define w  10)
(define rw 10)

(define (foo)    (set! rw (+  r  r)))
```

```
(define (bar)    (set! rw (- rw  r)))
(define (garply) (set! w  (+ rw rw)))
(define (buzz)   (set! w  (+  r  7)))
```

Notice that in these procedures `r` is what is known as a "read-only" variable because its value is only read from memory and never altered. `w` is a "write-only" variable because its value is only set, not examined. `rw` is both read and written.

For parts (a) and (b), we want to run the four procedures `foo`, `bar`, `garply`, and `buzz` in parallel, perhaps using serializers, with no other processes running.

(a) What is the minimum number of *distinct* serializers necessary to ensure that `r`, `rw`, and `w` all have correct values after our code executes? (Circle one.)

```
 0      1      2      3      4+
```

(b) Which (if any) of the procedures **do not** need to be serialized? **(Circle all that apply.)**

```
 foo      bar      garply      buzz
```

(c) What are the possible final values of `r`, `rw`, and `w` after *only* the following code executes (from the starting value 10 for all variables; note that `foo` isn't included):

```
(define protector (make-serializer))

(parallel-execute (protector bar)
                  (protector garply)
                  (protector buzz))
```

`r` values: ______________________________ `rw` values: ______________________________

`w` values: ______________________________

(d) What are the possible final values for `r`, `rw`, and `w` after *only* the following code executes (from the starting value 10 for all variables; note that `foo` isn't included):

```
(parallel-execute bar garply buzz)
```

`r` values: ______________________________ `rw` values: ______________________________

`w` values: ______________________________

We want to use object-oriented programming to simulate the effects of traffic in Berkeley. So far, we've designed an `automobile` class. This class has two instantiation variables: `fuel-efficiency` (in miles per gallon) and `fuel-capacity` (in gallons). Automobiles also have two methods. The first, `(drive m)`, causes the automobile to drive for m miles, or until it runs out of gas, whichever comes first. The second, `(fill-er-up)`, gives the automobile a full tank of gas. We'd like to add some more features to our simulation.

For each of the features below, identify the most appropriate way to add the feature to the simulation. **Also, give a one-sentence explanation of why you chose that answer.** Each feature may be one of the following:

- Child class (using the `parent` clause)

- Class variable
- Default method
- Initialize method/clause
- Instance variable
- Instantiation variable
- Method

`color`: Lets us know what color the automobile is.

`pollution`: Tells us how much pollution has been produced by all the automobiles of Berkeley, measured in smogallons. (One smogallon of pollution is produced whenever an automobile uses one gallon of gas.)

`miles-remaining`: Tells us how far this automobile can go on its current fuel.

`old-automobile`: Like a regular automobile, but produces two smogallons of pollution per gallon of fuel rather than one.

`starting-fuel`: New automobiles contain a random amount of fuel (between 0 and their capacity).

Starting `miles-remaining`: New automobiles also announce their `miles-remaining`.

The following expressions are typed, in sequence, at the Scheme prompt. Circle `#t` or `#f` to indicate the return values from the calls to `eq?`.

```
(define a (list 'x))
(define b (list 'x))
(define c (cons a b))
(define d (cons a b))

(eq? a b)                =>    #t   #f

(eq? (car a) (car b))    =>    #t   #f

(eq? (cdr a) (cdr b))    =>    #t   #f

(eq? c d)                =>    #t   #f

(eq? (cdr c) (cdr d))    =>    #t   #f

(define p a)
(set-car! p 'squeegee)
(eq? p a)                =>    #t   #f
```

```
(define q a)
(set-cdr! a q)
(eq? q a)                    =>    #t   #f

(define r a)
(set! r 'squeegee)
(eq? r a)                    =>    #t   #f
```

This week is the ASUC election. We're going to simulate the election in the adventure game by adding two new classes, `voting-place` and `student`.

A `voting-place` is just like a regular place, but when a student enters one for the first time, s/he is asked to vote. The voting places must maintain a list of candidates (the same list for every polling place) to present to the student, and keep a vote count for each of the candidates. (Hint: Use a table to hold the vote counts.)

At the beginning of the election, we send any voting place the message `start-election` with a list of candidates as argument:

```
> (define Sproul-Plaza (instantiate voting-place 'Sproul-Plaza))
> (ask sproul-plaza 'start-election '(Frankenstein Hoku Primm))
;; (Names are in alphabetical order -- no endorsement implied!)
```

This message should also initialize the vote counts to zero, and remember that no students have voted yet.

A `student` is just like a person, but with a `vote` method that takes a list of candidates as argument, and returns one of the candidates. (You can choose the candidate randomly or however you like.)

When a student who hasn't voted yet enters a voting place, the voting place should send the student a `vote` message with the list of candidates as argument. The return value will be one of the candidates, and the voting place should increase that candidate's vote count by one, and remember that this student has voted, so s/he won't be asked to vote again.

(Any object should accept a `student?` message, which should return true if the object is a student and false otherwise.)

Note: To simplify the problem, we are *not* asking for some features that would be necessary for a full simulation of elections, such as a method to find out who won!

(a) Create the `student` class.

(b) Now create the `voting-place` class.

Fill in the blanks with the response to the indicated expressions. The answers are 11, 121, 1001, and 1111 but not necessarily in that order!

(Hint: You shouldn't have to draw environment diagrams to figure this out. In each case, ask yourself: Is `A` a class variable or an instance variable? Is `B` a class variable or an instance variable?)

```
(define make-foo1
  (let ((a 1))
    (lambda ()
      (let ((b 1))
        (lambda ()
          (set! a (* a 10))
          (set! b (+ b a))
          b)))))


(define foo1-1 (make-foo1))
(define foo1-2 (make-foo1))
(foo1-1)
(foo1-1)
(foo1-2)  ==>  ____________________
```

```
(define make-foo2
  (let ((b 1))
    (lambda ()
      (let ((a 1))
        (lambda ()
          (set! a (* a 10))
          (set! b (+ b a))
          b)))))


(define foo2-1 (make-foo2))
(define foo2-2 (make-foo2))
(foo2-1)
(foo2-1)
(foo2-2)  ==>  ____________________
```

```
(define make-foo3
  (let ((a 1)
        (b 1))
    (lambda ()
      (lambda ()
        (set! a (* a 10))
        (set! b (+ b a))
        b))))


(define foo3-1 (make-foo3))
(define foo3-2 (make-foo3))
(foo3-1)
(foo3-1)
(foo3-2)  ==>  ____________________
```

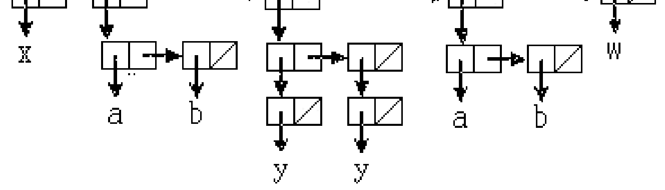
```
(define make-foo4
  (lambda ()
    (let ((a 1)
          (b 1))
      (lambda ()
        (set! a (* a 10))
        (set! b (+ b a))
        b))))


(define foo4-1 (make-foo4))
(define foo4-2 (make-foo4))
(foo4-1)
(foo4-1)
(foo4-2)  ==>  ____________________
```

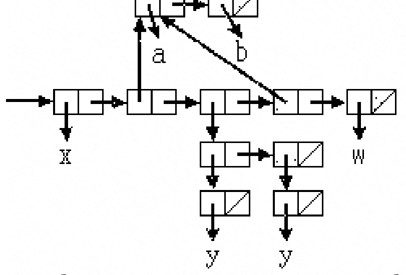
We would like to save memory in our data structures by ensuring that there are no duplicated top-level elements (that is, elements that are equal but not identical) in a list. For example, if we have the list

```
(x (a b) ((y) (y)) (a b) w)
```

then we'd like to modify the list so that the two copies of `(a b)` are not only `equal?` but also `eq?`. But we don't care about the two copies of `(y)`; they're not top-level elements.

Before and after pictures:





Complete the following definition of `make-eq!` so that it takes a list as its argument, and turns duplicate sublists into identical ones by mutation. It should return the modified list. The argument list after the procedure call should be `equal?` to the list before the procedure call, but possibly not `eq?`. **Do not allocate any new pairs!** Don't eliminate duplicated elements of elements, just top-level ones.

Note: `(member 'c '(a b c d))` returns `(c d)`, not `#t`.

```
(define (make-eq! lst)
  (if (not (pair lst))
      lst
      (let ((dup (member (car lst) (make-eq! (cdr lst)))))
```

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. If the expression results in an error, just say ERROR. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-car! (cddr x) (cadr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (car x) (cadr x))
  x)
(let ((x (list (list 1 2) (list 3 4))))
  (set-cdr! (cdr x) (cdar x))
  x)
```

Write a program `rotate!` that rotates the elements of a vector by one position. The function should alter the existing vector, not create a new one. It should return the vector. For example:

```
> (define v (make-vector 4))
> (vector-set! v 0 'a)
okay
> (vector-set! v 1 'b)
okay
> (vector-set! v 2 'c)
okay
> (vector-set! v 3 'd)
okay
> v
#(a b c d)
> (rotate! v)
#(d a b c)
```

(a) We would like to build a `client` class and a `server` class to simulate instant-messaging. **(Unlike the IM program you saw in lab, this is just a simulation; the clients and servers are all just objects in the same program on the same computer.)** Instances of the client class would connect to an instance of the server class, and the client objects would be able to send each other messages by way of the server.

A `server` has no instantiation variables. A `client` is instantiated with the name of the client and the `server` to connect to. Upon instantiation, it should connect to the given server.

You can ask a client for the most recent message that it has received. You can also ask a client to send a message by specifying the message to send and the name of the client to send to. For this problem, assume that there is a client by that name connected to the server (so you don't need to check this).

This is the desired interaction:

```
STk> (define s (instantiate server))
s
STk> (define brian (instantiate client 'brian s))
brian
STk> (define tyler (instantiate client 'tyler s))
tyler
STk> (ask tyler 'latest-message)
()
STk> (ask brian 'send-message '(where are the potstickers?) 'tyler)
okay
STk> (ask tyler 'latest-message)
(where are the potstickers?)
```

Fill in the blanks in the skeleton code below to implement the two classes. You do not have to use all the blanks. Do not write new methods or procedures.

```
(define-class (client name server)
  (instance-vars (latest-message '()))
  (initialize

    ______________________________________________________

    ______________________________________________________

    ______________________________________________________ )
  (method (receive-message msg)
    (set! latest-message msg))
  (method (send-message msg who)
    (ask server 'send-to-client msg who)))

(define-class (server)
  (instance-vars (clients '()))
  (method (accept-connection client)
    (set! clients (cons client clients)))
  (method (send-to-client msg name)

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________ ))
```

(b) Sometimes, we want to send a message to every client that's connected to the server. We write a subclass of the client class to allow this:

```
(define-class (broadcast-client name server)
  (parent (client name server))
  (method (broadcast msg)
    (ask server 'broadcast msg)))
```

The new `broadcast` method takes in a message to broadcast, and does not take in the name of the person to send to, since the message will be sent to everyone. The method simply calls the server object's broadcast method, which you will implement.

Also, every time a new client connects to the server, we would like the server to send a message to every client that someone new has joined us. The desired interaction is like this:

```
STk> (define bs (instantiate broadcast-server))
bs
STk> (define brian (instantiate broadcast-client 'brian bs))
brian
STk> (define tyler (instantiate broadcast-client 'tyler bs))
tyler
STk> (ask brian 'latest-message)
(tyler has joined us)
STk> (define chung (instantiate broadcast-client 'chung bs))
chung
STk> (ask brian 'latest-message)
(chung has joined us)
STk> (ask tyler 'latest-message)
(chung has joined us)
STk> (ask brian 'broadcast '(How is the midterm?))
okay
STk> (ask chung 'latest-message)
(how is the midterm?)
STk> (ask tyler 'latest-message)
(how is the midterm?)
STk> (ask chung 'send-message '(should be much harder) 'brian)
okay
STk> (ask brian 'latest-message)
(should be much harder)
STk> (ask tyler 'latest-message)
```

```
(how is the midterm?)
```

Fill in the blanks for the incomplete implementation of the `broadcast-server` class below that allows us to do this:

```
(define-class (broadcast-server name)
  (parent (server name))
  (method (accept-connection client)

    ________________________________________________

    ________________________________________________

    ________________________________________________
    (ask self 'broadcast
         (cons (ask client 'name) '(has joined us))))
  (method (broadcast msg)
    (for-each (lambda (c) (ask c 'receive-message msg))

             __________________________________________ )))
```

(a) Suppose we do this:

```
> (define x 3)

> (parallel-execute (lambda () (set! x 100))
                    (lambda () (set! x (+ x x))))
```

Assume that setting a variable to an integer value, looking up the value of a variable, and adding two integers are each atomic operations. What are all the possible values of `x` after the call to `parallel-execute`?

(b) What are all possible values of `x` after the call to `parallel-execute` if we do the following instead?

```
> (define x 3)

> (define x-protector (make-serializer))

> (parallel-execute (x-protector (lambda () (set! x 100)))
                    (x-protector (lambda () (set! x (+ x x)))))
```

Fill in the blanks to show what will be printed in response to the indicated expressions:

```
==> (define X '(a b c))
==> (define Y (cdr X))
==> (define Z (append Y Y))
==> (set-car! Y 123)
==> Y

________________________________________________
```

```
==> X
```

```
==> Z
```

```
==> (define L '(a))
==> (define M (append L L))
==> (set-car! (cdr M) 'w)
==> M
```

```
==> (define N (append! L L))
==> N
```

An *association list* is a list of pairs, such that the `car` of each pair is a *key* and the `cdr` of the pair is the corresponding *value*:

Write a procedure `invert` that takes an association list as its argument and modifies the list (without creating any new pairs) so that in each pair, what used to be the key is now the value and vice versa.

What will the Scheme interpreter print in response to each of the following expressions? (You need only show the response to the *last* expression in each group.) Also, draw a "box and pointer" diagram for the result of each expression:

```
(define x (list 1 '(2 3) 4))
(define y (cdr x))
(set-car! y 5)
x
```

```
(define a ((lambda (z) (cons z z)) (list 'a)))
(set-cdr! (car a) '(b))
a


(define r (list 'a 'b 'c))
(define s (list 'd 'e 'f))
(define p (append r s))
(set-car! (cdr p) 8)
r


(define e (list 1 2 3))
(set-cdr! (cdr e) nil)
e


(define f (list 1 2 3))
(set-car! (cdr f) nil)
f
```

Section 2.2.5 of the text considers the use of binary trees to allow fast $O(\log n)$ searching for a number in a set. (See pages 114–115.) At that time, we did not have the tool of mutable data available, and so their version of inserting a new element into the tree required them to recopy the entire tree. We would now like to modify this to insert new elements by mutation.

As in the case of tables, a *headed tree* (or htree) is a pair with `*tree*` as its car and a binary tree as its cdr.

A binary tree is either `the-empty-tree` or an entry value, left-branch and right-branch. (The book uses `nil` explicitly to represent the empty tree, but this is a violation of data abstraction.) The branches must themselves be trees, not numbers, even if its branches are empty. Here are the selectors and constructors:

```
(define the-empty-tree nil)

(define empty-tree? null?)

(define (entry tree) (car tree))

(define (left-branch tree) (cadr tree))

(define (right-branch tree) (caddr tree))

(define (make-tree entry left right)
  (list entry left right))

(define (make-empty-headed-tree) (cons '*tree* the-empty-tree))
```

(a) Write `(set-left-branch! tree x)`, which takes an unheaded tree and sets its left branch to be `x`. Also write `set-right-branch!` and `set-entry!`.

(b) Fill in the gaps in the following procedure for inserting a new element.

```
(define (hinsert! x htree)
  (if (empty-tree? (cdr htree))
      (____________________________ htree
                   (make-tree x the-empty-tree the-empty-tree))
      (insert! x (cdr htree))))

(define (insert! x tree)
  (cond ((< x (entry tree))
         (if (empty-tree? (left-branch tree))
             (set-left-branch! tree
                               (make-tree ____________________________))
             ____________________________))
        ((> x (entry tree))
         (if (empty-tree? (right-branch tree))
             (set-right-branch! tree
                               (make-tree ____________________________))
             ____________________________))
        (else nil) ))
```

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression:

```
(let ((x (list 1 2 3)))
  (set-cdr! (cdr x) 4)
  x)


(let ((x (list 1 2 3)))
  (set! (car x) 4)
  x)


(let ((x (list 1 2 3)))
  (set-car! (cdr x) (cddr x))
  x)


(let ((x (list 1 2 3)))
  (set-cdr! (cdr x) x)
  x)
```

Write a procedure named `map!` that takes a function and a list, like the ordinary `map`, but instead of returning a new list it should modify the argument list so that each element is replaced with the result of applying the function to that element.

At home I have a Radio Shack remote control system that allows me to control lights and appliances from in bed. Next to my bed is a *controller* that has 16 on-off switches for individual devices, plus a switch that can send the signals "all lights on" and "all units off." (It is a safety feature that the all-on signal only affects lights and not appliances.) To make this work, I have to replace wall switches (for lights) and outlet boxes (for appliances) with special *units* that have a unit number from 1 to 16.

When I push the "unit 3 on" button on the controller, the controller sends the message `on 3` to every unit. (The signal is broadcast through the power wiring in my apartment; there isn't a separate connection from the controller to each unit. That `on 3` message is ignored by units other than those whose unit number is 3.

Here is an implementation of the controller:

```
(define-class (controller)
  (instance-vars (units '()))
  (method (attach unit) (set! units (cons unit units)))
  (method (on n) (for-each (lambda (u) (ask u 'on n)) units))
  (method (off n) (for-each (lambda (u) (ask u 'off n)) units))
  (method (all-lights-on)
    (for-each (lambda (u) (ask u 'all-on)) units))
  (method (all-units-off)
    (for-each (lambda (u) (ask u 'all-off)) units)) )
```

(a) Implement a light-unit object that is created with a unit number, that has a local variable called `power` whose value is initially `#f` but is changed to `#t` when the light is on, and that accepts the four messages sent out by the controller object.

(b) Implement an appliance-unit object that inherits from light-unit. It should work exactly the same except that it ignores the `all-on` message.

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression:

```
(let ((x (list 1 2 3)))
  (set-cdr! (cdr x) (cddr x))
  x)
(let ((x (list 1 2 3)))
  (set! x (cdr x))
  x)
(let ((x (list 1 2 3)))
  (set-car! x (cddr x))
  x)
(let ((x (list 1 2 3)))
  (set-car! (cdr x) (cdr x))
  x)
```

For the purposes of the next two questions, you may assume that the variable `deck-of-cards` contains a list of 52 playing cards:

```
(define deck-of-cards '(ah 2h 3h ... 10h jh qh kh as 2s ... qc kc))
```

where the word `ah` represents the ace of hearts, and so on up to `kc` for the king of clubs.

Given below is a function that shuffles a deck by mutation, keeping the same pairs in the resulting list, in the same order, but rearranging which element is stored in which pair.

```
(define (shuffle! deck)
  (define (shuffle-helper! deck count)
    (if (= count 0)
        '()
        (let ((k (random count)))
          (swap! deck (nthcdr k deck))
          (shuffle-helper! (cdr deck) (- count 1)))))
  (shuffle-helper! deck (length deck))
  deck)
```

The expression `(nthcdr k deck)` should return the result of removing the first `k` cards from the `deck`.

The call `(swap! deck1 deck2)` interchanges the first card of `deck1` and the first card of `deck2`, using mutation.

(a) Write the procedures `nthcdr` and `swap!`.

(b) How would the effect and the returned value change if the word "`deck`" were removed from the last line of the definition of the `shuffle!` procedure?

(a) A deck object responds to two messages: `deal` and `empty?`. It responds to `deal` by returning the top card of the deck, after removing that card from the deck; if the deck is empty, it responds to `deal` by returning `()`. It responds to `empty?` by returning `#t` or `#f`, according to whether all cards have been dealt.

Write a procedure `make-deck` that returns an object representing a shuffled deck of 52 cards. Use `make-object` and `object-cond` as appropriate. You may use the original `shuffle` procedure from the previous question without rewriting it here.

(b) Suppose the first shuffling of a deck returns a list that begins `(2c 4d as ...)` Explain the result of

```
(ask make-deck 'deal)
```

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3)))
  (set-car! x (cdr x))
  x)
(let ((x (list 1 2 3)))
  (set-cdr! (cdr x) (car x))
  x)
```

We are going to add magic wands to the adventure game. A magic wand is a kind of `thing` that you can possess. When you wave the wand, it transports you instantly to some particular place. For example:

```
> (define wand-1 (instantiate wand 'silver shin-shin))
> (ask evans 'appear wand-1)
> (ask brian 'go 'down)
BRIAN MOVED FROM CSUA-OFFICE TO EVANS
> (ask brian 'take wand-1)
BRIAN TOOK SILVER-WAND
TAKEN
> (ask brian 'wave wand-1)
BRIAN MOVED FROM EVANS TO SHIN-SHIN
```

We need to invent the `wand` class and to add a `wave` method to the `person` class. Here is the `wave` method:

```
(define-class (person name place)
  ...
  (method (wave wand)
    (if (member wand possessions)
        (ask wand 'magic)
        (error "You don't own that wand!") ))
  ...)
```

Wands must understand a `magic` message; when a wand gets this message, it asks its possessor to `go-directly-to` its predetermined place.

Write the `wand` class definition.

One of the problems in the object system is that an object can't make direct use of its parent's instance variables. For example:

```
(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (1+ count))
    count) )

(define-class (incrementer)
  (parent (counter))
  (method (next amount)
    (set! count (+ count amount))       ; This won't work!
    count) )
```

On the next page is an environment diagram showing the (simplified) result of defining these classes, and creating an instance

```
(define upper (instantiate incrementer))
```

(a) Extend the diagram on the next page to show the result of

```
((upper 'next) 3)                        ; Note two invocations.
```

[This expression is equivalent to the OOP `(ask upper 'next 3)`.]

(b) Briefly explain, with reference to the diagram, why the `set!` in the `incrementer` class doesn't do what you want.

(diagram here)

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-cdr! (cdr x) (cdddr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cdr x) (cadr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-cdr! (cdr x) (car x))
  x)
(let ((x (list 1 (2 3) 4)))
  (set-car! x (cadr x))
  x)
```

Write `deep-subst!`, a procedure that takes three arguments, two of which are words and the third of which is any list structure (anything made of pairs). It should mutate the list structure so that any occurrence of the first argument, however deep in sublists, is replaced by the second argument. Examples:

```
> (deep-subst! 'foo 'baz (list (cons 'hello 'goodbye) (cons 'moby 'foo)))
((hello . goodbye) (moby . baz))

> (deep-subst! 'a 'x (list (list 'a 'b 'c) (list 'b 'a 'd)
                           (list 'f 'a 'b)))
((x b c) (b x d) (f x b))
```

Do not allocate any new pairs in your solution!

Question 3 (5 points):

We are simulating a read-only memory (ROM) in the OOP system:

```
(define-class (rom values)
  (default-method
    (if (and (number? message) (< message (length values)))
        (nth message values)
        (error "Bad ROM address"))))

> (define sample-rom (instantiate rom '(4 5 9 23))
> (ask sample-rom 2)
9
> (ask sample-rom 7)
ERROR -- Bad ROM address
```

Now we want to make a programmable ROM (PROM). Unlike a standard ROM, a PROM starts with nothing stored in it, although it does have a fixed size that is set when the PROM is built. You can put a value into each address, but only once—you can't change the value later. We want to implement the prom class so that it takes the size as its instantiation variable, but inherits from the rom class:

```
(define (prom size)
  (parent (rom ((repeated (lambda (vals) (cons '() vals)) size)
                '())))
  ...)
```

It should accept a SET message that takes an address (a number) and a value (anything) as arguments. If the prom is big enough to have such an address, but the value in that address is the empty list, then the new value should be put in that address.

```
> (define this-prom (instantiate prom 6))
> (ask this-prom 3)
()
> (ask this-prom 'set 3 'foo)
> (ask this-prom 3)
FOO
> (ask this-prom 'set 9 'baz)
ERROR -- Bad ROM address
> (ask this-prom 'set 3 'garply)
ERROR -- PROM address already has value
```

Complete the definition of the prom class. If possible, do it without modifying the definition of the rom class. If you must modify the rom definition, explain why.

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-cdr! x (caddr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cdr x) (cddr x))
  x)
(let ((x (list 1 (list 2 3) 4)))
  (set-car! (cadr x) (car x))
  x)
```

Define an object class called `password-protect`. The purpose of the class is to allow an object to be "hidden" so that a password is needed to send it messages. Here's how it works. Suppose we have this class definition:

```
(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (+ count 1))
    count))
```

In order to make a password-protected counter, we want to be able to do this:

```
> (define ppc (instantiate password-protect
                           (instantiate counter) 'exotic))
PPC
> (ask ppc 'next)
ERROR: Password incorrect
> (ask (ask ppc 'exotic) 'next)
1
> (ask (ask ppc 'exotic) 'next)
2
```

In this example, `exotic` is the password for the protected counter. When sent this password as a message, the object `ppc` returns the underlying counter object, which can then be sent its own messages.

Write `deep-map!`, a procedure that takes an arbitrary list structure, applies a given function to each leaf, and *modifies the argument list* to replace each leaf with the value returned by the function. For example:

```
> (define x (list (list 3 4) 5 (list) (list (list 6))))
x
> x
((3 4) 5 () ((6)))
> (deep-map! square x)
((9 16) 25 () ((36)))
> x
((9 16) 25 () ((36)))
```

For the purposes of this problem, a "leaf" is anything that isn't a pair or the empty list.

Do not allocate any new pairs in your solution! Modify the existing list structure. (You are not, of course, responsible for any pairs that might be allocated by the function you are given as argument, like `square` in the example above.)

What will the Scheme interpreter print in response to each of the following expressions?

Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3 4)))
  (set-cdr! (cddr x) (car x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cddr x) (cddddr x))
  x)
(let ((x (list 1 2 3 4)))
  (set-car! (cddr x) x)
  x)
```

Write `list-rotate!` which takes two arguments, a nonnegative integer `n` and a list `seq`. It returns a mutated version of the argument list, in which the first `n` elements are moved to the end of the list, like this:

```
> (list-rotate! 3 (list 'a 'b 'c 'd 'e 'f 'g))
(d e f g a b c)
```

You may assume that $0 \leq$ `n` $<$ `(length seq)` without error checking.

Note: **Do not allocate any new pairs** in your solution. Rearrange the existing pairs.

(a) Suppose that `bh` is a person object, in the place `bh-office`. What does each of these do?

```
(ask (ask bh 'place) 'name)
(ask (ask bh 'name) 'place)
```

(b) Here are some situations that might be simulated using oop. In each case we want to know whether class A should be a `parent` of class B (answer Yes or No):

- We're simulating a rock and roll group. Class A: musician. Class B: drummer.
- We're simulating an automobile. Class A: automobile. Class B: wheel.
- We're simulating an office. Class A: file cabinet. Class B: file folder.

(c) For each of the following, should it be a **class** variable or an **instance** variable?

- In the file cabinet class, the number of files in a file cabinet.
- In the AC Transit local bus class, the price of a bus ticket.
- In the restaurant class in the adventure game, how many people have eaten at this restaurant.

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list (list 1 2) (list 3 4))))
  (set-car! (car x) (cadr x))
  x)
(let ((x (list (list 1 2) (list 3 4))))
  (set-car! (cdr x) (cdar x))
  x)
(let ((x (list (list 1 2) (list 3 4))))
  (set-cdr! (cdar x) (cadr x))
  x)
```

This problem is about binary trees, in which the nodes are represented in the form indicated by these selectors and constructor:

```
(define datum car)
(define left-branch cadr)
(define right-branch cddr)

(define (make-tree datum left right)
  (cons datum (cons left right)))
```

The empty tree is represented by the empty list.

(a) Complete the implementation of this abstract data type by writing the three mutators for binary tree nodes.

(b) Binary search trees provide efficient searching only if they are well-balanced, with about as many nodes in the left branch as in the right branch (at every level). There are many algorithms for allowing new data to be inserted into a binary search tree while ensuring that the tree remains balanced. Some of those algorithms involve a technique called *rotation*, in which some elements of the tree are rearranged while preserving the binary search tree order requirements. The pictures below show the general idea (with triangles representing subtrees) and a specific example, in which the subtree whose root datum is 15 is rotated.

Write the procedure `rotate!` that takes a tree node as its argument and rotates the tree rooted at that node, by mutation. The pair at the head of the given tree (that is, the pair you are given as the argument) must still be the head of the resulting tree, because some higher-up tree may point to that pair.

Note: **Do not allocate any new pairs** in your solution. Modify the existing pairs. Also, **respect the data abstraction.**

Consider the following OOP class:

```
(define-class (foo value)
  (class-vars (foos '()))
  (initialize (set! foos (cons self foos))))
```

Don't forget that the OOP system provides methods `value` and `foos` that return the values of the corresponding variables.

Your job is to implement a similar behavior in ordinary Scheme, by defining a procedure `make-foo` that works as shown in the following example:

```
> (define f1 (make-foo 3))
> (define f2 (make-foo 4))
> (f1 'value)
3
> (f2 'value)
4
> (f1 'foos)
(<procedure> <procedure>)   ; however Scheme prints procedures f2 and f1
> (map (lambda (foo) (foo 'value))
       (f1 'foos))
(4 3)
```

Note that we invoke `f1` and `f2` directly; you are not using `ask` or `instantiate` from the OOP language. Note also that all objects created by `make-foo` will give the same answer to the message `foos`.

In the book, `make-serializer` is implemented using a mutex. `Make-mutex` is implemented using the atomic `test-and-set!` operation, like this:

```
(define (make-mutex)   ; from SICP page 312
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire)))  ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))
```

Instead, suppose that you are given serializers as a primitive capability; write `make-mutex` using serializers (and *not* using `test-and-set!`) to provide concurrency control.