Question 1 (3 points):

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)
```

Question 2 (4 points):

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!**

Question 3 (4 points):

Define a stream named `all-integers` that includes all the integers: positive, negative, and zero.

You may use any stream or procedure defined in the text.

Question 4 (4 points):

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.

On the following pages is some relevant code from the adventure game project.

Group Question (4 points):

Draw the environment diagram for the situation after the following definition and invocation have been evaluated:

```
(define (every-nth num lst)
  (define (help lst pos)
    (cond ((null? lst) '())
          ((= (remainder pos num) 0)
           (cons (car lst) (help (cdr lst) (1+ pos))))
          (else (help (cdr lst) (1+ pos)))))
  (help lst 1))

(every-nth 2 '(she loves you))
```

```
(define-class (place name)
  (parent (basic-object))
  (instance-vars
    (directions-and-neighbors '())
    (things '())
    (entry-procs '())
    (exit-procs '()))
  (method (place?) #t)
  (method (type) 'place)
  (method (neighbors) (mapcar cdr directions-and-neighbors))
  (method (exits) (mapcar car directions-and-neighbors))
  (method (look-in direction)
    (let ((pair (associate direction directions-and-neighbors)))
      (if (not pair)
          nil                     ;; nothing in that direction
          (cdr pair))))           ;; return the place object
  (method (appear new-thing)
    (if (memq new-thing things)
        (error "Thing already in this place" (list name new-thing)))
    (set! things (cons new-thing things))
    'appeared)
  (method (gone thing)
    (if (not (memq thing things))
        (error "Disappearing thing not here" (list name thing)))
    (set! things (delete thing things))
    'disappeared)

  (method (new-neighbor direction neighbor)
    (if (associate direction directions-and-neighbors)
        (error "Direction already assigned a neighbor" (list name direction)))
    (set! directions-and-neighbors
        (cons (cons direction neighbor) directions-and-neighbors))
    'connected)

  (method (enter who)
    (mapcar (lambda (proc) (proc)) entry-procs)
    (forall (delete who (filter things person?))
            (lambda (pers) (ask pers 'notice))))
  (method (exit) (forall exit-procs  (lambda (proc) (proc))))
  (method (add-entry-procedure proc)
    (set! entry-procs (cons proc entry-procs)))
  (method (add-exit-procedure proc)
    (set! exit-procs (cons proc exit-procs)))
  (method (remove-entry-procedure proc)
    (set! entry-procs (delete proc entry-procs)))
  (method (remove-exit-procedure proc)
    (set! exit-procs (delete proc exit-procs)))
  (method (clear-all-procs)
    (set! exit-procs nil)
    (set! entry-procs nil)
    'cleared))
```

```
(define-class (person name place)
  (parent (basic-object))
  (instance-vars
    (possessions '())
    (saying "")
    (money 100))
  (initialize
    (ask place 'appear self)
    (ask self 'put 'strength 1000))
  (method (person?) #t)
  (method (type) 'person)
  (method (look-around)
    (mapcar (lambda (obj) (ask obj 'name))
            (filter (ask place 'things)
                    (lambda (thing)
                      (not (eq? thing self))))))
  (method (take thing)
    (cond ((not (thing? thing)) (error "not a thing" thing))
          ((not (memq thing (ask place 'things)))
           (error "thing taken not at this place"
                  (list (ask place 'name) thing)))
          ((memq thing possessions) (error "you already have it!"))
          ((or (eq? (ask thing 'possessor) 'no-one)
               (ask (ask thing 'possessor) 'may-take? self thing))
           (announce-take name thing)

           ;; add it to my possessions
           (set! possessions (cons thing possessions))

           ;; go through all the people at the place
           ;; if they have the object we are taking
           ;; make them lose it and have a fit
           (forall
            (filter (ask place 'things) person?)
            (lambda (pers)
              (if (and (not (eq? pers self)) ; ignore myself
                       (memq thing (ask pers 'possessions)))
                  (sequence
                   (ask pers 'lose thing)
                   (have-fit pers)))))

           ;; actually change the possessor
           (ask thing 'change-possessor self)
           'taken)
          (else (announce-too-weak name thing) )))

  (method (may-take? who what)
    (if (> (ask who 'strength) (ask self 'strength))
        what
        #f))

  (method (take-all)
    (forall (filter (ask place 'things)
                    (lambda (thing)
                      (and (thing? thing)
                           (eq? (ask thing 'possessor) 'no-one))) )
            (lambda (thing) (ask self 'take thing)) ))
  (method (lose thing)
    (set! possessions (delete thing possessions))
    (ask thing 'change-possessor 'no-one)
    'lost)

  (method (eat)
    (forall (filter possessions edible?)
            (lambda (food)
              (ask self 'put 'strength (+ (ask self 'strength)
                                          (ask food 'calories) ))
              (ask self 'lose food)
              (ask place 'gone food))))

  (method (get-money amt)
    (set! money (+ money amt)) )
  (method (pay-money amt)
    (cond ((>= money amt)
           (set! money (- money amt))
           #t)
          (else #f) ))

  (method (buy food-name)
    (let ((food (ask place 'sell self food-name)))
      (if food
          (sequence
           (set! possessions (cons food possessions))
           (ask food 'change-possessor self))
          (error "could not buy" food-name) )))

  (method (talk) (print saying))
  (method (set-talk string) (set! saying string))
  (method (exits) (ask place 'exits))
  (method (notice) (ask self 'talk))
  (method (go direction)
    (let ((new-place (ask place 'look-in direction)))
      (cond ((null? new-place)
             (error "Can't go" direction))
            (else
             (ask place 'exit)
             (announce-move name place new-place)
             (forall possessions
                     (lambda (p)
                       (ask place 'gone p)
                       (ask new-place 'appear p)))
             (ask place 'gone self)
             (ask new-place 'appear self)
             (set! place new-place)
             (ask new-place 'enter self)))))

  (method (go-directly-to new-place)
    (announce-move name place new-place)
    (forall possessions
            (lambda (p)
              (ask place 'gone p)
              (ask new-place 'appear p)))
    (ask place 'gone self)
    (ask new-place 'appear self)
    (set! place new-place)))
```