Problem 1.

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

(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)

(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) '())
e


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

(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)

(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)

(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)

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

(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)

(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)

(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)

(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 1 2 3 4)))
  (set-cdr! (cdr x) (cdddr 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! (cddr x) (cddddr x))
  x)

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

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

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

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

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

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

(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)
```

Object-Oriented Programming

Problem 2.

```
(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))`

Problem 3.

(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.

Problem 4.

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`.

Problem 5.

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
```

Problem 6.

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.

Problem 7.

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))))
```

Problem 8.

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) ...)
  ...)
```

Problem 9.

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.

Problem 10.

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.

Problem 11.

(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.

Assignment, State, and Environments

Problem 12.

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))))
```

Problem 13.

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)))))
```

Problem 14.

In this problem you're going to write a piece of a simplified Adventure game, not using our OOP notation, just in regular Scheme. We are concentrating on the behavior of people, so a place will just be represented as a room number. 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!

Problem 15.

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   ______No

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.

Problem 16.

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


(define foo1-1 (make-foo1))        (define foo3-1 (make-foo3))
```

```
(define foo1-2 (make-foo1))        (define foo3-2 (make-foo3))
(foo1-1)                           (foo3-1)
(foo1-1)                           (foo3-1)
(foo1-2)  ==>   _______            (foo3-2)  ==>   _______


(define make-foo2                  (define make-foo4
  (let ((b 1))                       (lambda ()
    (lambda ()                         (let ((a 1)
      (let ((a 1))                           (b 1))
        (lambda ()                       (lambda ()
          (set! a (* a 10))                (set! a (* a 10))
          (set! b (+ b a))                 (set! b (+ b a))
          b)))))                           b))))


(define foo2-1 (make-foo2))        (define foo4-1 (make-foo4))
(define foo2-2 (make-foo2))        (define foo4-2 (make-foo4))
(foo2-1)                           (foo4-1)
(foo2-1)                           (foo4-1)
(foo2-2)  ==>   _______            (foo4-2)  ==>   _______
```

Problem 17.

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.

Problem 18.

(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?

Problem 19.

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) ))))
```

Problem 20.

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.

[For review purposes, we are not providing the diagrams – just draw a diagram for each of the examples!]

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)
```

Drawing environment diagrams

Problem 21.

Draw the environment diagram resulting from evaluating the following expressions, and show the result printed by the last expression where indicated.

```
> (define (bar x)
    (let ((z (lambda (b) (* x b)))
          (c x))
      (lambda (x)
        (set! c (z (* c x)))
        c)))

> (define foo (bar 4))

> (foo 3)
```

____

Problem 22.

Draw the environment diagram that results from the following interactions, and fill in the blank with the value printed:

```
>(define a 8)

>(define b 9)

> (let ((a 3)
        (f (lambda (b) (+ a b))))
    (f 5))
```

____

Problem 23.

Draw the environment diagram resulting from evaluating the following expressions, and show the results printed by the expressions where indicated.

```
> (define make-foo
    (let ((y 4))
      (lambda ()
        (set! y (* y 2))
        (let ((z y))
          (lambda ()
            (set! z (+ z 1))
            z)))))

> (define foo1 (make-foo))
```

```
> (define foo2 (make-foo))

> (foo1)
```

____

```
> (foo2)
```

____

Problem 24.

Draw the environment diagram resulting from evaluating the following expressions, and show the result printed by the last expression where indicated.

```
> (define x 4)

> (define (baz x)
    (define (* a b) (+ a b))
    (lambda (y) (* x y)))

> (define foo (baz (* 3 10)))

> (foo (* 2 x))
```

____

Problem 25.

Here are some Scheme expressions. Fill in the blanks to indicate the appropriate return values and draw the final environment diagram.

```
> (define y 5)

> (define (agent x)
    (let ((y 0))
      (lambda ()
        (x)
         y) ) )

> (define mission
    (agent (lambda () (set! y (+ y 1)))) )

> (mission)
> (mission)
```

```
> y
```

Problem 26.

Draw the environment diagram resulting from evaluating the following expressions, and show the result printed by the last expression where indicated.

```
> (define x 2)

> (define (f x)
    (if (even? x)
        (f (- x 1))
        (lambda (y) (+ x y))))

> (define g (f 4))

> (g 15)
```

Problem 27.

Draw the environment diagram that results from the following interactions, and fill in the blanks with the values printed:

```
> (define (inc var)
    (set! var (+ var 1)))

> (define x 5)

> (define foo
    (let ((x 7)
          (z 100))
      (lambda (y)
        (inc x)
        (set! z (* z x))
```

```
        (+ z y))))

> (foo 10)
```

```
> x
```

```
> (foo 20)
```

Problem 28.

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

```
(define maximizer
  (let ((value 0))
    (lambda (arg)
      (if (> arg value)
          (set! value arg))
      value)))

(define foo (maximizer 6))
```

Problem 29.

Draw the environment diagram resulting from evaluating the following expressions, and show the result printed by the last expression where indicated.

```
> (define foo
    (lambda (x f)
      (if f
          (f 7)
          (foo 5 (lambda (y) (+ x y))))))

> (foo 3 #f)
```

List mutation

Problem 30.

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.

Problem 31.

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

a (1 point).

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

b (2 points).

```
(define (funny! x)
  (if (null? x)
      '()
      (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
```

Problem 32.

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 '()))
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!`.

Problem 33.

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!**

Problem 34.

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.

Problem 35.

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
```

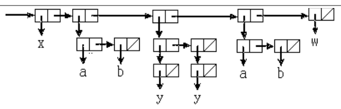
Problem 36.

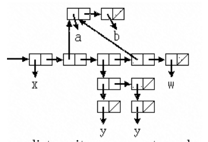
We would like to save memory in our data structures by ensuring that there are no duplicated top-level elements (that is, elements that are equal but not identical) in a list. For example, if we have the list

```
(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)))))
```

Problem 37.

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!

Problem 38.

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.)

Problem 39.

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.**

Vectors

Problem 40.

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.

Problem 41.

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)
```

Concurrency

Problem 42.

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)

Problem 43.

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.

Problem 44.

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)))))
```

Problem 45.

(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?

Problem 46.

(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?

Problem 47.

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?

Problem 48.

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: ____________________

Problem 49.

(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)))))
```

Problem 50.

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.