**Question 1 (4 points):**

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

```
(cons (append '(a b) '(c d)) '(e f))

(list (cons 'a 'b) (list 'c 'd) 'e)

(caaddr '((a b c d e) (f g h i j) (l m n o p) (q r s t u)))

(cons 'a (cdr '(((b)))))
```

**Question 2 (5 points):**

Recall that the definition of a *list* is that it's either the empty list or a pair whose cdr is a list. This definition says nothing about the elements of the list. For example,

```
(1 2 (3 . 4) (5 . 6) 7)
```

is a list of five elements, even though two of the elements are pairs that aren't lists.

We'll define a *deep list* as a list in which every element is either a deep list or not a pair. Equivalently, a deep list is a structure made of pairs in which the cdr of every pair, including the ones in the elements, is a list.

Write the predicate `deep-list?` that takes any Scheme object as its argument, returning `#t` if the argument is a deep list.

**Question 3 (5 points):**

We want to write a program that uses the time of day as an abstract data type. We'll represent times internally as a list of three elements, such as `(11 23 am)` for 11:23 am. For the purposes of this problem, assume that the hour part is never 12, so there's never any special problems about noon and midnight. The hour will be a number 1–11, the minute will be a number 0–59, and the third element (which we'll call the *category*) must be the word `am` or the word `pm`. Here's our implementation:

```
(define (make-time hr mn cat) (list hr mn cat))
(define hour car)
(define minute cadr)
(define category caddr)
```

(a) This is a good internal representation, but not a good representation for the user of our program to see. Write a function `time-print-form` that takes a time as its argument and returns a word of the form `3:07pm`.

(b) If we want to ask whether one time is before or after another, it's convenient to use the 24-hour representation in which 3:47 pm has the form `1547`. Write a procedure `24-hour` that takes a time as its argument and returns the number that represents that time in 24-hour notation:

```
> (24-hour (make-time 3 47 'pm))
1547
```

Respect the data abstraction!

(c) Now we decide to change the *internal* representation of times to be a number in 24-hour form. But we want the constructor and selectors to have the same interface so that programs using the abstract data type don't have to change. Rewrite the constructor and selectors to accomplish this.

**Question 4 (5 points):**

The example of message passing on page 141 deals only with functions of one argument, such as the magnitude of a complex number. In this problem our goal is to extend the idea of message passing to two-argument functions like addition. We want to do this without violating the spirit of "intelligent data objects"; that is, a complex number should know how to add itself to another one.

We define the two-argument functions in the following form:

```
(define (+complex z1 z2)
  ((z1 'add) z2))
```

Your job is to modify `make-rectangular` so that complex numbers understand this new message. For your convenience here is the code from the book:

```
(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-RECTANGULAR" m))))
  dispatch)
```