Sample midterm 2 #3

Problem 1 (What will Scheme print?)

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write “error”; you don’t have to provide the exact text of the message. Also, draw a box and pointer diagram for the value produced by each expression.

(append (list 'a 'b) '(c d))

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

(list (list 'a 'b) (append '(c) '(d)))

(cdar '((1 2) (3 4)))

Problem 2 (Tree recursion)

The following definition comes from page 116 of SICP:

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

In this problem you’re going to extend the idea of accumulation from sequences to “deep lists”: lists of lists of lists to arbitrary depth. Write deep-accumulate, a function of three arguments: a two-argument function, an initial value, and a list structure. It should work like this:

> (deep-accumulate + 0 '(3 (4 (5) ((6) 7) 8) (9 10)))
52
Problem 3 (Tree recursion)

Write the function `datum-filter` which, given a predicate and a Tree, returns a list of all the datums in the Tree that satisfy the predicate (in any order). We are using general Trees (trees that can have any number of children) as defined in lecture. We are not using binary trees. The function should return the empty list for any Tree in which no datums satisfy the predicate. You may use helper procedures.

For example:
```
(datum-filter even? (5))
```
```
      /
     / \
    (12) (19)
   /  |
  /   |
 /    |
   4   |
    /  |
   (4) (22) (27)
```

returns the list `(12 4 22)` in any order.
Problem 4 (Message passing)

Consider the following message-passing implementation of rational numbers:

(define (make-rat num den)
  (define (dispatch msg)
    (cond ((eq? msg 'numer) num)
          ((eq? msg 'denom) den)
          (else (error "Unknown op -- MAKE-RAT" msg)) ))
  dispatch)

Compare this with the complex number implementation on page 186 of the text. Suppose we are given analogous procedures make-integer and make-real. You are going to modify make-rat, and write some auxiliary procedures, to achieve three goals:

(a) In the conventional and data-directed styles of programming, we could take an arbitrary number and find out what type it is, by using the type-tag function. This function would return a symbol such as rational or complex to identify the type of its argument. In the message-passing implementation, a number is no longer a pair whose car is the type name. We want a type function that will allow us to find out the type of a message-passing-style number.

(b) For homework you created a raise operation that could be applied to a number to convert it to the next higher type in the tower of types. You did this in the context of data-directed programming, using a table. We want to implement this raise operation in message-passing style. In particular, we want to allow a rational number to be raised to type real.

(c) Once we have the idea of raising numbers, it makes sense that we should be able to ask the same questions about a given number that we could ask about higher types. For example, we’d like

(\(\text{real-part (make-rat 3 4)}\) \(\Rightarrow\) 0.75
(\(\text{imag-part (make-rat 3 4)}\) \(\Rightarrow\) 0

But we don’t want to have to re-implement all the messages for real numbers and complex numbers within make-rat. Instead, if make-rat doesn’t recognize a message, it should raise the number and send the message to the raised number.

Rewrite make-rat to meet these three goals, and write the procedures type (for goal a) and raise (for goal b).
Problem 5 (Object oriented programming)

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.