1. (6 points) Suppose the following Racket program has been read in.

```racket
(define x '(1 2 3))
(define y '(10 11 12))
(define z (append (cdr x) (cdr y)))
```

Draw box-and-arrow diagram of the resulting lists, being careful to show correctly what parts are shared and what parts are separate. (If you run out of room, please draw a fresh copy of the diagram on the back of this page instead.)
2. (12 points) Consider the following Haskell function definitions. (Some of these are from the Prelude; some are written from scratch.)

\[
\begin{align*}
\text{filter } p \ p x s &= [ x | x <- x s, p x] \\
\text{repeat } x &= x : \text{repeat } x \\
\text{memb } x s a &= \text{or} (\text{map } (=a) x s) \\
\text{vowel } &= \text{memb } "aeiou"
\end{align*}
\]

Here are some possible types for each of these functions. After each type, write G if the type is correct and the most general, C if it is correct but not the most general, and I if it is incorrect.

\[
\begin{align*}
\text{filter } &: \ [a] -> [a] -> [a] \\
\text{filter } &: \ (a -> \text{Bool}) -> [a] -> [a] \\
\text{filter } &: \ (\text{Eq } a) => (a -> \text{Bool}) -> [a] -> [a] \\
\text{filter } &: \ (\text{Num } a) => (a -> \text{Bool}) -> [a] -> [a] \\
\text{repeat } &: \ a -> a \\
\text{repeat } &: \ a -> [a] \\
\text{repeat } &: \ [a] -> [[a]] \\
\text{memb } &: \ [a] -> a -> \text{Bool} \\
\text{memb } &: \ \text{Eq } a => [a] -> a -> \text{Bool} \\
\text{memb } &: \ \text{Ord } a => [a] -> a -> \text{Bool} \\
\text{vowel } &: \ \text{Char} -> \text{Bool} \\
\text{vowel } &: \ \text{String} -> \text{Bool}
\end{align*}
\]

3. (10 points) Consider a Haskell function \texttt{remove} that takes an item and a list, and returns a new list, dropping elements that are equal to the item. For example, \texttt{remove 3 [1,2,3,4,3,2,1]} evaluates to \texttt{[1,2,4,2,1]}, and \texttt{remove 3 []} evaluates to \texttt{[]}.

(a) What is the most general type for \texttt{remove}?

(b) Write a recursive definition of \texttt{remove}.

(c) Write a non-recursive definition of \texttt{remove} using Haskell's \texttt{filter} function. (Hint: there is a definition of \texttt{filter} in Question 2.)
4. (5 points) What is the value of mystery? (If it’s infinite give the first several elements.)

\[ \text{mystery} = 0 : \text{map} (\lambda x \to 2 \cdot x + 1) \text{ mystery} \]

5. (6 points) Define a curried plus function \texttt{curried-plus} in Racket. So \texttt{(curried-plus 1)} should return a function that adds 1 to numbers, and \texttt{((curried-plus 1) 3)} should evaluate to 4.

6. (6 points) Now define a function \texttt{extra-spicy-plus} in Racket that is also curried but that works with 3 numbers instead of 2. So \texttt{(((extra-spicy-plus 1) 2) 3)} should evaluate to 6.

7. (6 points) Consider the following Racket expressions. (They are identical except that the first uses \texttt{let}, the second uses \texttt{let*}, and the third uses \texttt{letrec}.) What does each one evaluate to?

\[
\begin{align*}
\text{(let \ ([y 1]}
& [f (lambda (x) (+ x 1))])
\text{ (let \ ([y 5]}
& [f (lambda (x) (if (> x 5) x (f (+ x y))))])
\text{ (f 3))})
\end{align*}
\]

\[
\begin{align*}
\text{(let \ ([y 1]}
& [f (lambda (x) (+ x 1))])
\text{ (let* \ ([y 5]}
& [f (lambda (x) (if (> x 5) x (f (+ x y))))])
\text{ (f 3))})
\end{align*}
\]

\[
\begin{align*}
\text{(let \ ([y 1]}
& [f (lambda (x) (+ x 1))])
\text{ (letrec \ ([y 5]}
& [f (lambda (x) (if (> x 5) x (f (+ x y))))])
\text{ (f 3))})
\end{align*}
\]
8. (5 points) Consider the following Racket program.

\[
\begin{align*}
\text{(define } &\ x\ 1) \\
\text{(define } &\ z\ 2) \\
\text{(define (add-it } &\ y) \ (\ +\ x\ y)) \\
\text{(define (test)} \\
\text{(let ([x 10]} \\
\text{[z 20]} \\
\text{)} \ (\text{add-it}\ z)))
\end{align*}
\]

(a) What is the result of evaluating \text{(test)}? 

(b) Suppose Racket used dynamic scoping. What would be the result of evaluating \text{(test)}? 

9. (8 points) The lecture notes for Racket macros included a cosmetic macro for \texttt{my-if} (which just provides a different syntax for \texttt{if}).

\[
\begin{align*}
\text{(define-syntax my-if} &\ ; \text{macro name} \\
\text{(syntax-rules (then else)} &\ ; \text{literals it uses, if any} \\
\text{[(my-if} &\ e1\ \text{then e2 else e3)} &\ ; \text{pattern} \\
\text{(if} &\ e1\ e2\ e3)))) &\ ; \text{template}
\end{align*}
\]

Show the code to add \texttt{my-if} to OCTOPUS. For full credit, do this by rewriting the \texttt{my-if} expression as a normal \texttt{if} and then calling \texttt{eval} on the result, in the same way that the Racket \texttt{my-if} macro produces new Racket source code that is then interpreted.

Hint: you should write another case for the \texttt{eval} function. Here are two example cases from the starter file:

-- A quoted expression evaluates to that expression.
\texttt{eval (OctoList [OctoSymbol "quote", x]) env = x}

{- An expression starting with \texttt{(lambda ...)} evaluates to a closure,
where a closure consists of a list of variable names \texttt{(OctoSymbols),}
the environment of definition, and the body. -}
\texttt{eval (OctoList [OctoSymbol "lambda", OctoList vars, body]) env =
OctoClosure vars env body}
10. (8 points) Write a slightly silly Racket macro called `const` that does something like `const` in Haskell: the result of evaluating `(const exp1 exp2)` should be the result of evaluating `exp1` (just throw `exp2` away without evaluating it). Unlike Haskell, though, evaluate `exp1` each time the `const` expression is evaluated. For example, `(const (+ 2 3) (/ 1 0))` should evaluate to 5. (There is no divide-by-zero error since we don’t evaluate the second expression.)

11. (8 points) Convert the following Haskell action into an equivalent one that doesn’t use `do`.

```haskell
echo = do
  x <- readLn
  y <- readLn
  putStrLn "the sum is \\
  putStrLn (show (x y))
```