CSE 341 - Programming Languages
Midterm - Autumn 2015

Your Name:

```
(for recording grades):
 1. (max 6)

 2. (max 12)

 3. (max 10)

 4. (max 5)

 5. (max 6)

 6. (max 6)

 7. (max 6)

 8. (max 5)

 9. (max 8)

10. (max 8)

11. (max 8)
```

```
Total  (max 80):
```

You can bring a maximum of 2 (paper) pages of notes. No laptops, tablets, or smart phones.

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

```
(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.)

```
filter p xs = [ x | x <- xs, p x]
repeat x = x : repeat x
memb xs a = or (map (==a) xs)
vowel = memb "aeiou"
```

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.

```
filter :: [a] -> [a] -> [a]

filter :: (a -> Bool) -> [a] -> [a]

filter :: (Eq a) => (a -> Bool) -> [a] -> [a]

filter :: (Num a) => (a -> Bool) -> [a] -> [a]

repeat :: a -> a

repeat :: a -> [a]

repeat :: [a] -> [[a]]

memb :: [a] -> a -> Bool

memb :: Eq a => [a] -> a -> Bool

memb :: Ord a => [a] -> a -> Bool

vowel :: Char -> Bool

vowel :: String -> Bool
```

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

   (a) What is the most general type for `remove`?

   (b) Write a recursive definition of `remove`.

   (c) Write a non-recursive definition of `remove` using Haskell's `filter` function. (Hint: there is a definition of `filter` in Question 2.)

4. (5 points) What is the value of `mystery`? (If it's infinite give the first several elements.)

```
mystery = 0 : map (\x->2*x+1) mystery
```

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

6. (6 points) Now define a function `extra-spicy-plus` in Racket that is also curried but that works with 3 numbers instead of 2. So `(((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 `let`, the second uses `let*`, and the third uses `letrec`.) What does each one evaluate to?

```
(let ([y 1]
      [f (lambda (x) (+ x 1))])
  (let ([y 5]
        [f (lambda (x) (if (> x 5) x (f (+ x y))))])
    (f 3)))
```

```
(let ([y 1]
      [f (lambda (x) (+ x 1))])
  (let* ([y 5]
         [f (lambda (x) (if (> x 5) x (f (+ x y))))])
    (f 3)))
```

```
(let ([y 1]
      [f (lambda (x) (+ x 1))])
  (letrec ([y 5]
           [f (lambda (x) (if (> x 5) x (f (+ x y))))])
    (f 3)))
```

8. (5 points) Consider the following Racket program.

```
(define x 1)
(define z 2)

(define (add-it y)
   (+ x y))

(define (test)
  (let ([x 10]
        [z 20])
    (add-it z)))
```

(a) What is the result of evaluating `(test)`?

(b) Suppose Racket used dynamic scoping. What would be the result of evaluating `(test)`?

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

```
(define-syntax my-if              ; macro name
  (syntax-rules (then else)       ; literals it uses, if any
    [(my-if e1 then e2 else e3)   ; pattern
     (if e1 e2 e3)]))             ; template
```

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

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

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

{- An expression starting with (lambda ...) evaluates to a closure,
where a closure consists of a list of variable names (OctoSymbols),
the environment of definition, and the body. -}
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.

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