CSE 341 - Programming Languages
Midterm - Autumn 2012 - Answer Key

1. (8 points) Suppose that we have a `filter` function in Haskell, defined as follows:

```
filter f [] = []
filter f (x:xs) | f x       = x : filter f xs
                | otherwise = filter f xs
```

For example, `filter even [1,2,3,4]` evaluates to `[2,4]`.

Circle each type declaration that is a correct type for `filter`. (Not necessarily the most general type, just a correct one.)

```
filter :: (Int -> Bool) -> [Int] -> [Int]  CORRECT

filter :: a -> [a] -> [a]  WRONG

filter :: (a -> b) -> [a] -> [b]  WRONG

filter :: (a -> Bool) -> [a] -> [a] CORRECT AND MOST GENERAL

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

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

Which of the above types, if any, is the most general type for `filter`?

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

2. (5 points) What are the first 6 elements in the following Haskell infinite list?

```
mystery = 0 : (map (\x -> 10-x) mystery)

[0,10,0,10,0,10]
```

3. (4 points) What are the free variables in each of the following Racket expressions?

```
(lambda (x) (if (= x 1) (+ y z) 100))
y z

(let ((x 5) (y x)) (+ x y))
x

(lambda (a) (+ b (let ((b 10)) (+ a b))))
b

(let ((s '(1 2 3))) (map f s))
f
```

4. (6 points) Convert the following Haskell action into an equivalent one that doesn't use do.

```
echo = do
    putStr "type in a number between 1 and 10: "
    n <- readLn
    putStrLn ("Double your number is " ++ show (2*n))

echo2 = putStr "type in a number between 1 and 10: " >> readLn >>=
    \n -> putStrLn ("Double your number is " ++ show (2*n))
```

5. (12 points) Consider the function `rss` that finds the square root of the sum of the squares of a list of numbers.

   (a) Write a Racket definition of `rss`. You can define a helper function if needed, and you can use built-in Racket functions like `sqrt` and `map`. However, Racket doesn't have a built-in function to find the sum of a list of numbers.

```
;; using the apply trick to define sum .... also ok to define this as
;; a recursive helper function
(define (rss s)
  (let* ((squares (map (lambda (x) (* x x)) s))
         (sum (apply + squares)))
    (sqrt sum)))
```

   (b) Write a point-free Haskell definition of `rss`. Again, you can use built-in Haskell functions, which conveniently include `sum` as well as `sqrt` and `map`. Also give the most general type for this function.
   Hint: the type of `sqrt` is
      `Floating a => a -> a`
   where `Floating` is a subclass of `Num`. Partial credit for a correct but non-point-free answer.

```
rss = sqrt . sum . map (^2)

rss :: Floating c => [c] -> c

also OK to give the type:

rss :: [Double] -> Double

Haskell infers the more general type for the type of
sqrt . sum . map (^2)

but the more specific type for rss -- but if you give it the more
general type for rss it will accept it and that will be the type.

This is similar to Haskell's behavior with the type of 0

and of
x = 0
```

6. (8 points) Define an `average` function in Racket that finds the average of two numbers.

```
(define (average1 x y)
  (/ (+ x y) 2))
```

Now define a curried version of `average` in Racket.

```
(define (average2 x)
  (lambda (y) (/ (+ x y) 2)))

;; or a short version:
(define ((average3 x) y)
  (/ (+ x y) 2))
```

7. (6 points) What is the result of evaluating the following Racket expressions?

```
(let ((x 1)
      (y 2)
      (z 5))
  (let* ((x (+ x 10))
         (y x))
    (+ x y z)))
27

(let ((x 1)
      (y 2)
      (z 5))
  (let ((x (+ x 10))
        (y x))
    (+ x y z)))
17

(let* ((x 10)
       (f (lambda (y) (* x y)))
       (x 20))
  (f x))
200
```

8. (6 points) Consider the following MUPL program.

```
(eval-prog
 (mlet "x" (int 100)
       (mlet "plus1"
             (fun #f "x" (add (var "x") (int 1)))
             (call (var "plus1") (add (var "x") (var "x"))))))
```

For the following questions, write out the environments as lists, in exactly the form used by the MUPL interpreter. If you need to refer to the closure for `plus1` in your answers, you can abbreviate it as `<plus1 closure>` and receive full credit, although it's fine to write it out completely if you prefer.

(For partial credit, write down the environment in some other understandable form.)

(a) What is the environment bound in the closure for the `plus1` function?

```
(list (cons "x" (int 100)))
```

(b) What is the environment that MUPL uses when evaluating the body of the `plus1` function when it is called in the above program?

```
(list (cons "x" (int 200)) (cons "x" (int 100)))
```

(c) What is the environment that MUPL uses when evaluating the actual parameter to the call to the `plus1` function?

```
(list (cons "plus1" <plus1 closure>) (cons "x" (int 100)))
```

or written out:

```
(list
 (cons "plus1" (closure (list (cons "x" (int 100)))
                        (fun #f "x" (add (var "x") (int 1)))))
 (cons "x" (int 100)))
```

9. (5 points) Suppose the following Racket code is evaluated. How many times is `squid` printed? How many times is `octopus` printed?

```
(define d1 (delay (print "squid ") (+ 3 4)))
(define d2 (delay (print "octopus ") (+ 8 8)))
(force d1)
(force d1)

squid: 1
octopus: 0
```