CSE 341 - Programming Languages
Midterm - Winter 2010 - Answer Key

1. (8 points) Suppose that we have a `merge` function in Haskell that takes two sorted lists, and merges them to produce a new sorted list. Duplicate elements are deleted. (This is the same function that was used in the Hamming number example, but with additional cases so that it works on both finite and infinite lists.) Here's its definition:

```
merge xs [] = xs
merge [] ys = ys
merge (x:a) (y:b) | x<y   = x : merge a (y:b)
                  | x==y  = y : merge a b
                  | x>y   = y : merge (x:a) b
```

For example, `merge [2,5,8] [1,5]` evaluates to `[1,2,5,8]`.

These are correct types for `merge`:

```
merge :: [Int] -> [Int] -> [Int]

merge :: (Ord t) => [t] -> [t] -> [t]
```

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

```
merge :: (Ord t) => [t] -> [t] -> [t]
```

2. (10 points) Suppose that we want to rewrite the Scheme symbolic differentiation program in Haskell. Haskell doesn't have the nice program/data equivalence that Scheme does, but we can still define an appropriate expression data type in Haskell. Complete the following program by adding `basic_deriv` and `simplify` cases for `Times`. After you've added these, for example `deriv (Times (Const 3) (Var "x")) "x"` should evaluate to `Const 3`.

```
data Expr = Var String | Const Int | Plus Expr Expr | Times Expr Expr
              deriving (Show,Read)

-- the derivative of an expression with respect to a variable
deriv :: Expr -> String -> Expr
deriv exp x = simplify (basic_deriv exp x)

basic_deriv :: Expr -> String -> Expr

basic_deriv (Const c) x = Const 0
basic_deriv (Var v) x | v==x = Const 1
                      | otherwise = Const 0

basic_deriv (Plus e1 e2) x = Plus (basic_deriv e1 x) (basic_deriv e2 x)

-- *** NEW ***
basic_deriv (Times e1 e2) x =
    let d1 = Times e1 (basic_deriv e2 x)
        d2 = Times (basic_deriv e1 x) e2
    in Plus d1 d2
```

```
simplify :: Expr -> Expr

-- no further simplification possible for Consts and Vars
simplify (Const c) = Const c
simplify (Var v) = Var v

simplify (Plus e1 e2) =
    let s1 = simplify e1
        s2 = simplify e2
    in simplify_plus s1 s2

-- *** NEW ***
simplify (Times e1 e2) =
    let s1 = simplify e1
        s2 = simplify e2
    in simplify_times s1 s2

simplify_plus (Const c1) (Const c2) = Const (c1+c2)
simplify_plus (Const 0) e = e
simplify_plus e (Const 0) = e
simplify_plus e1 e2 = Plus e1 e2

-- *** NEW ***
simplify_times (Const c1) (Const c2) = Const (c1*c2)
simplify_times (Const 0) e = (Const 0)
simplify_times e (Const 0) = (Const 0)
simplify_times (Const 1) e = e
simplify_times e (Const 1) = e
simplify_times e1 e2 = Times e1 e2
```

3. (5 points) What are the first 8 elements in the following list?

```
mystery = 0 : 10 : (map (+1) mystery)

[0,10,1,11,2,12,3,13]
```

4. (10 points) Convert the following Haskell action into an equivalent one that doesn't use `do`.

```
echo = do
    putStr "type in a sentence: "
    s <- getLine
    putStrLn ("You typed " ++ s)

echo2 = putStr "type in a sentence: " >> getLine >>=
        \s -> putStrLn ("You typed " ++ s)
```

5. (6 points) Find the squid! For each of the following variables, write an expression that picks out the symbol `squid`. For example, for this definition: `(define w '(clam squid octopus))` the answer is `(cadr w)` (or `(car (cdr w))` is OK as well).

(a) `(define x '(clam octopus starfish squid))`
`(cadddr x)`

(b) `(define y '(clam ((oyster squid)) mollusc))`
`(car (cdaadr y))`

(c) `(define z '(((squid))))`
`(caaar z)`

6. (8 points) Suppose the following Scheme definitions have been read in:

```
(define (myappend xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (myappend (cdr xs) ys))))

(define a '(squid clam))
(define b '(octopus mollusc geoduck))
(define c (myappend a b))
```

What is the result of evaluating each of the following expressions?

(a) `(eq? (car a) (car c))`
`#t`

(b) `(eq? (cdr a) (cdr c))`
`#f`

(c) `(eq? b (cddr c))`
`#t`

(d)
```
(let ((a '(whale))
        (b a))
   (myappend a b))

(whale squid clam)
```

7. (10 points) Write a Scheme function `noduplicates` that takes a list and returns a new list that drops any duplicate elements. The new list can be in any order. You can use the built-in Scheme function `member` if you would like. Here are some examples. (The value returned by your function should contain the same elements but can be in a different order.)

```
(noduplicates '(a b c d c d d))  =>  (a b c d)
(noduplicates '(a b c ))  =>  (a b c)
(noduplicates '())  =>  ()

(define (noduplicates s)
  (cond ((null? s) '())
        ((member (car s) (cdr s)) (noduplicates (cdr s)))
        (else (cons (car s) (noduplicates (cdr s))))))
```

8. (6 points) Parameter passing. Consider the following program in an Algol-like language.

```
begin
integer n;
procedure p(k: integer);
    begin
    k := 2*k;
    n := n+k;
    print(n);
    end;
n := 10;
p(n);
print(n);
end;
```

What is printed if k is passed by value?
30 30

What is printed if k is passed by value-result?
30 20

What is printed if k is passed by reference?
40 40

9. (12 points) Tacky but easy-to-grade true/false questions!

   (a) Suppose in Haskell we have a variable `j` of type `Int` and a variable `x` of type `Float`. When compiling code for the expression `j+x`, Haskell automatically coerces `j` to be a `Float`. False.

   (b) If Haskell passed parameters by name rather than using lazy evaluation, the only effect would be on performance — there would never be programs that used to work and that now give errors. True.

   (c) If Haskell passed parameters by value rather than using lazy evaluation, the only effect would be on performance — there would never be programs that used to work and that now give errors. False.

   (d) In Scheme, arguments are passed by reference. False.

   (e) In Scheme, `and` and `or` can't be implemented as ordinary Scheme functions, since we don't always want to evaluate all of the arguments. True.

   (f) In Haskell, the keyword `do` is used to include a number of Haskell statements in a do loop, and provides an alternative to recursion for performing iterations. False.