CSE 341 - Programming Languages
Midterm - Winter 2009 - Answer Key

Open book and notes. No laptop computers, PDAs, internet-equipped cellphones, or similar devices. (Calculators are OK, although you won't need one.) Please answer the problems on the exam paper — if you need extra space use the back of a page.

60 points total

1. (8 points) Suppose that we have a `duplicate` function in Haskell that takes a number `n` and an item `x`, and returns a list with `n` occurrences of `x`. Here's its definition:

```
duplicate 0 x = []
duplicate n x = x : duplicate (n-1) x
```

These are correct types for `duplicate`. (Not necessarily the most general type, just a correct one.)

```
duplicate :: Integer -> Integer -> [Integer]

duplicate :: (Num a) => a -> b -> [b]
```

These aren't correct types:

```
duplicate :: Bool -> Bool -> [Bool]

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

duplicate :: (Ord a) => a -> b -> [b]

duplicate :: a -> b -> [b]
```

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

```
duplicate :: (Num a) => a -> b -> [b]
```

2. (10 points) Suppose the following Haskell program has been read in.

```
my_sum [] = 0
my_sum (x:xs) = x + my_sum xs
count x ys = my_sum (map (\y -> if x==y then 1 else 0) ys)

read_bool = do
  b <- readLn
  return (not b)
```

What is the *value* of each of the following expressions? (Some may give a type error; if so say that.)

(a) `my_sum [10,30,50]  =>  90`

(b) `my_sum (10,30)  =>  type error`

(c) `count 'e' "The octopus ate the clam"  => 3`

(d) `count True [1,2,3,4]  =>  type error`

What is the most general *type* of each of the following expressions? Some of them may give type errors — if so, say that.

(a) `my_sum :: (Num t) => [t] -> t`

(b) `count :: (Eq a, Num t) => a -> [a] -> t`

(c) `count 'x' :: (Num t) => [Char] -> t`

(d) `read_bool :: IO Bool`

(e) `not read_bool  => type error`

(f) `putStrLn "enter True or False: " >> read_bool >>=     \n -> putStrLn (show n) :: IO`

3. (5 points) Is the `my_sum` function in Question 2 tail recursive? If not, write a tail recursive version (in Haskell still). You can write a helper function if needed.

It is not recursive. Here is a tail recursive version, using a helper function:

```
my_sum s = sum_helper s 0

sum_helper [] total = total
sum_helper (x:xs) total = sum_helper xs (x+total)
```

4. (5 points) What are the first 6 elements in the following list?

```
mystery = 1 : 2 : (map (*10) mystery)
```

```
[1,2,10,20,100,200]
```

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 '(squid clam octopus))` the answer is `(car w)`.

   (a) `(define x '(clam octopus squid starfish))  =>  (caddr x)`

   (b) `(define y '((octopus squid) mollusc))  =>  (cadar y)`

   (c) `(define z '(octopus . squid))  =>  (cdr z)`

6. (10 points) Write a Scheme function `count` that takes two values: `x` and `y`. Assume that `x` is a symbol. If `y` is a list, `count` returns the number of occurrences of `x` in the list. However, unlike the Haskell version in Question 2, the Scheme version can take lists of lists of lists — you need to recursively descend into the structure as far as possible to count the x's. You can assume the list doesn't have any cycles. If `y` isn't a list, return 1 if `x` is `eq` to `y`, and otherwise 0. For example:

```
(count 'c '(a b c d (a b c) (((a c)))))    => 3
(count 'x '())  => 0
(count 'x '(a b c))  => 0
(count 'x 'x)  => 1
(count 'x 'y)  => 0
```

```
(define (count x ys)
  (cond ((pair? ys) (+ (count x (car ys)) (count x (cdr ys))))
        ((eq? x ys) 1)
        (else 0)))
```

7. (8 points) Tacky but easy-to-grade true/false questions!

   (a) A hygenic macro gives fresh names to local variables at each use of the macro, to avoid name collisions. True.

   (b) A hygenic macro flosses and brushes daily. False. (Although this is kind of a silly question, which might trip up non-native speakers of English, so we didn't count off for "True.")

   (c) One definition of the term "strongly typed" equates it with "statically typed." Under this definition, Haskell is strongly typed but Scheme is not. True.

   (d) Another definition of the term "strongly typed" equates it with "type safe." Under this definition, Scheme is strongly typed but Haskell is not. False.

8. (8 points) Consider a dynamically typed version of Haskell, called D-Haskell. Everything else about D-Haskell is the same as in regular Haskell.

   Are there any programs that give type errors in Haskell but that don't give type errors in D-Haskell? If so give an example. Are there any programs that pass Haskell's type checker and that give a runtime error; but that don't give a runtime error in D-Haskell?

   There are programs that give type errors in Haskell but that don't give type errors in D-Haskell, namely programs with a type error in an expression that is never evaluated. Here is an example using the built-in function `const`, which doesn't evaluate its second argument:

```
const 3 ([] + [1,2])
```

   This gives a type error in Haskell but not in D-Haskell.

   There aren't any programs that pass Haskell's type checker and that give a runtime error; but that don't give a runtime error in D-Haskell.