CS 457/557 Homework 6 – due 2pm, Tuesday, November 15, 2005

Hand in all your solutions on paper *and* email the solutions to `cs457acc@cs.pdx.edu`. All the solutions should be placed in a single `.hs` file, which should be an attachment.

1. Do Hudak exercise 12.2, using the 7-color version of `Color` on p. 160. Hint: You can save yourself a quadratic amount of typing by looking carefully at the *full* definitions of the `Ord` and `Enum` classes given in Ch. 24. Warning: You do *not* get a static error message if you fail to define all the needed members in a class instance declaration – though you will get a politely-phrased runtime error if you try to use an undefined member.

2. Suppose we define a new class, inspired by the Java `Serialiazable` interface, describing types that can be "flattened" into a compact sequence of bytes (suitable for storing or transmitting over a network connection) and subsequently can be recovered.

```
import Word                -- unsigned integers of various sizes
type Bytestream = [Word8]
class Serializable a where
  serialize :: a -> Bytestream
  deserialize :: Bytestream -> (a,Bytestream)
```

Here `serialize` x produces a `Bytestream` containing an encoding of x, and `deserialize` does the inverse: it converts a prefix of its argument to a value of type `a` and returns that value together with the remainder of the bytestream. We might use them as follows:

```
a :: Char
a = 'a'
b :: [Int]
b = [101,104]
c :: Maybe String
c = Just "pdq"
bytes ::  Bytestream
bytes = serialize a ++ serialize b ++ serialize c

-- and later on...
a':: Char
(a',rest1) = deserialize bytes
b' :: [Int]
(b',rest2) = deserialize rest1
c' :: Maybe String
(c',_) = deserialize rest2
-- now should have a == a', b == b', c == c'
```

Complete the following instance declarations. Assume that `Int` is 32 bits wide. You'll find the `Bits` library useful.

```
instance Serializable Char where ...
instance Serializable Int where ...
instance Serializable a => Serializable (Maybe a) where ...
instance Serializable a => Serializable [a] where ...
```

3. (from Fasel and Hudak, "A Gentle Introduction to Haskell," SIGPLAN Notices 27(5), May 1992.)

Consider the following general statement about object-oriented programming languages like C++ or Java:

> *Classes* capture common sets of *operations*. A particular *object* may be an *instance* of a class, and will have a *method* corresponding to each operation. Classes may be arranged hierarchically, forming notions of *superclasses* and *subclasses*, and permitting *inheritance* of operations/methods. A *default method* may also be associated with an operation.

If we substitute "*type class*" for "class" and "*type*" for "object" this statement yields a valid summary of Haskell's type class mechanism.

Yet type classes do *not* really support object-oriented programming. Why not? Discuss, and give examples of object-oriented idioms that cannot be written conveniently in Haskell.

(Note: This is obviously an open-ended question. Think seriously about it for awhile, and then try to write up a **brief** answer.)