Programming I: Functional Programming in Haskell
Solutions to Unassessed Exercises

Set 6: Type classes

1. (a) `minBound :: Colour`

   (b) i. `succ Green`

   ii. `fromEnum Blue`

   iii. `toEnum 0 :: Colour` – note that you need to force the type, otherwise Haskell has no way to know that you want a `Colour`, rather than a `Bool`, for example

   iv. `enumFromTo Red Blue`

2. We'll make the 24-hour format constructor take two integers (hours and minutes), but you could equally define it to take one. If you do this, however, you'll need to do `div` and `mod` base 100, to extract the hours and minutes components of the integer.

```
data AmPm = AM | PM

data Time = TwentyFour Int Int |
            WallClock Int Int AmPm
          deriving (Eq)  -- Delete this for part (c)
```

(a)
```
to24 :: Time -> Time
to24 (WallClock h m AM)
  | h == 12   = TwentyFour 0 m
  | otherwise = TwentyFour h m
to24 (WallClock h m PM)
  | h == 12   = TwentyFour h m
  | otherwise = TwentyFour (h + 12) m
to24 t
  = t
```

(b)
```
equalTime :: Time -> Time -> Bool
equalTime t1 t2
  = isSame (to24 t1) (to24 t2)
  where
    isSame (TwentyFour h m) (TwentyFour h' m')
      = h == h' && m == m'
```

(c)
```
instance Eq Time where
  (==) = equalTime
```

(d)
```
instance Show Time where
  show (WallClock 12 0 AM)   = "Midday"
  show (WallClock 12 0 PM)   = "Midnight"
  show (WallClock h m AM)    = show' h ++ ":" ++ show' m ++ "AM"
  show (WallClock h m PM)    = show' h ++ ":" ++ show' m ++ "PM"
  show (TwentyFour h m) = show' h ++ show' m ++ "HRS"
```

```
show' :: Int -> String
show' n
  | n < 10    = "0" ++ show n
  | otherwise = show n
```

3. Something like the following will do. This uses `fromJust` from `Data.Maybe`.

```
instance Vars Exp where
  x = Id "x"
  y = Id "y"
  z = Id "z"

-- This requires the FlexibleInstances language extension...
instance Vars String where
  x = "x"
  y = "y"
  z = "z"

lift :: Fun -> Exp -> Exp -> Exp
lift f x y = App f x y

instance Num Exp where
  fromInteger = Val . fromInteger
  (+) = lift Add
  (-) = lift Sub
  (*) = lift Mul

lookUp :: Eq a => a -> [(a, b)] -> b
lookUp
  = (fromJust .) . lookup

update :: VarName -> Int -> Environment Int -> Environment Int
update v x env
  = (v, x) : [p | p@(v', x') <- env, v /= v']

eval :: Exp -> Environment Int -> Int
eval (Val n) env
  = n
eval (Id  id) env
  = lookUp id env
eval (App f e e') env
  = apply f (eval e env) (eval e' env)

apply op v v'
  = lookUp op [(Add, (+)),(Sub, (-)), (Mul, (*))] v v'

evalS :: Statement -> Environment Int -> Environment Int
evalS (A (v, e)) env
  = update v (eval e env) env
evalS (Loop n p) env
  = foldr run' env (replicate n p)

run :: Program -> Environment Int
```

```
run p
  = run' p []
  where
    run' p env
      = foldr evalS env (reverse p)
```