**1. What are the types of the following x1, x2, … x5? Some might have type errors:**

```
b = True

x1 :: IO ()
x1 = if b then putStrLn "ho" else return ()

-- Type Error
x2 = if b then putStrLn "squid"  else return "octopus"

x3 :: [Char]
x3 = if b then "squid" else "octopus"

-- Type error
x4 = if b then "squid" else return ()

x5 :: IO Bool
x5 = do
     putStr "testing"
     x <- readLn
     return (not x)
```

**2. Give a recursive definition of a list** `doubles` **whose first element is 10, and whose n th element is twice the n− 1 st, i.e., [10, 20, 40, 80, 160, 320, ....]. To do this, write a helper function doubles_from that takes a parameter n and returns a list of all the doubles starting at n.**

```
doubles :: [Integer]
doubles = doubles_from 10
doubles_from :: Integer -> [Integer]
doubles_from n = n : doubles_from (2*n)

-- other version of doubles
doubles2 :: [Integer]
doubles2 = 10 : map (*2) doubles
```

**3. Give yet another non-recursive definition of** `doubles` **using the built-in function** `iterate` **from the Haskell prelude. This is defined as follows:**

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

doubles3 :: [Integer]
doubles3 = iterate (*2) 10
```

**4. Define a Haskell list** `dollars` **that is the infinite list of amounts of money you have every year, assuming you start with $100 and get paid 5% interest, compounded yearly. (Ignore inflation, deflation, taxes, bailouts, the possibility of total economic collapse, and other such details.) So dollars should be equal to [100.0, 105.0, 110.25, ...]**

```
-- simple but not general version:
dollars :: [Double]
dollars = 100 : map (\d -> 1.05*d) dollars

-- or using iterate:
idollars = iterate (1.05*) 100

-- more general recursive version:
better_dollars :: [Double]
better_dollars = dollar_growth 100.0 0.05

dollar_growth :: Double -> Double -> [Double]
dollar_growth p rate = p : dollar_growth (p*(1+rate)) rate
```

**5. Desugar the following actions:**

```
lion_desugar =
  putStrLn "What is the color of your mane?"
  >> getLine
  >>= \x -> putStrLn $ "Rawr, nice " ++ x ++ " mane"


parity_repl_desugar =
  putStrLn "Enter a number"
  >> readLn
  >>= \n -> case odd n of
            True -> putStrLn $ (show n) ++ " is odd"
            False -> putStrLn $ (show n) ++ " is even"
  >> parity_repl_desugar

map_reduce_desugar =
  putStrLn "Enter a unary mapping operation"
  >> getLine
  >>= \op -> putStrLn "Enter a unary reducing operation"
             >> getLine
             >>= \reduce -> putStrLn "Enter a list to evaluate"
                            >> getLine
                            >>= \lst -> let expr = "foldr1 (" ++
reduce ++ ") $ map (" ++ op ++ ") " ++ lst
                                        in evaluate expr
```

**Key Takeaways from Octopus discussion:**

You can pattern match in a number of ways, so long as it's a valid pattern match.

If a function is passed in [OctoInt 5, OctoList [OctoInt 6]], I can pattern match on this with (x : xs), [x, y], [x, OctoList y], (x : y : ys), (x : (OctoList y) : ys), and more!

One way to help reveal what the functions you're given take in is to define the following (using octocons as an example):
octocons args = error ("args is " ++ show args)

In other words, just throw an error! You can start from here and make the way you're pattern matching more precise as you move forward.