```haskell
--import Text.Printf
--printf :: PrintfType r => String -> r
-- ([Typeclass] [var]) says that [var] must be in a certain typeclass in order to proceed. You can also have more than one restriction. (Int a, Show a, Read a)
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"


-- PATTERN MATCHING --

-- When you call sayMe, it goes down the list of functions and specific arguments. You need to be careful
-- in that you make sure that you have a catch for every input that's possible so that something does get returned.
-- The last function declaration should catch anything and everything.
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5."


-- It can fail if you have a pattern that can't be matched. We don't have a catch-all at the end here.
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"


-- Function to add together vectors
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

-- Functions to get the 1st, 2nd, and 3rd value out of a tripple.
first :: (a, a, a) -> a
first (a, _, _) = a
second :: (a, a, a) -> a
second (_, a, _) = a
third :: (a, a, a) -> a
third (_, _, a) = a


-- Head function to find the first value of a list. We use the _ to say that we don't really care about the value in that place.
head' :: [a] -> [a]
head' [] = error "Can't call head on an empty list."
head' (x:_:z:_) = [z, x]
head' (x:_) = [x]

-- Function to tell us something about a list that is passed. We use x:y:z because that says you're getting the first, second, and third elements from a list
-- that is passed. We could also say [x,y], but that would prevent us from using a catch-all at the end because we can't write an infinite amount of lists.
-- [] tells us that we are looking at an empty list.
tell :: (Show a) => [a] -> String
tell [] = "This is an empty list."
tell (x:[]) = "This list has one item: " ++ show x ++ "."
tell (x:y:[]) = "This list has two items: " ++ show x ++ " and " ++ show y ++ "."
tell (x:y:_) = "This list has more than two items. The first two are " ++ show x ++ " and " ++ show y ++ "."

-- Patterns (@) are a neat way of getting a value, but keeping the original values
capital :: String -> String
capital "" = "Empty string."
capital all@(x:_) = "The first letter of '" ++ all ++ "' is " ++ [x] ++ "."



-- RECURSION --

-- Here, you're calling the factorial funciton, but you call it back onto itself until it gets to factorial 0, where it goes to 1, because we have that defined as 1.
-- If we didn't define factorial 0 = 1, then the function would never terminate.
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)


-- Length function. Takes a list of some sort and gives a number back. Takes the first element from the list and calls the length' function on the rest of it recursively.
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:x) = 1 + length' x


-- Sum function to find the sum of all of the values in a list. The values of the lsit must be in the Num typeclass.
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs



-- GUARDS --

-- Guards are a way of testing if something is true or false based on a condition. If one guard condition fails, then it falls through to the next one. "otherwise" is
-- defined as True always.
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
        | weight / height ^ 2 <= 18.5 = "You're underweight you emo you."
        | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
        | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
        | otherwise                   = "You're a whale, congratulations!"


-- Maximum function to tell us the maximum of two arguments.
max' :: (Ord a) => a -> a -> a
max' a b
        | a > b     = a
        | otherwise = b


-- Comparison function to tell the relation of one things toanother.
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b -- We can also define functions using backticks. It is sometimes easier to read them that way.
        | a > b     = GT
        | a < b     = LT
        | otherwise = EQ



-- WHERE --

-- You can use "where" to do a calculation or evaluation once. More efficient than using a calculation every time. Then whole function can see them.
bmiTell' :: (RealFloat a, Show a) => a -> a -> String
bmiTell' weight height
```

```haskell
        | bmi <= skinny = "You're underweight you emo you. Your BMI is " ++ show bmi
        | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly! Your BMI is " ++ show bmi
        | bmi <= fat    = "You're fat! Lose some weight, fatty! Your BMI is " ++ show bmi
        | otherwise     = "You're a whale, congratulations! Your BMI is " ++ show bmi
        where
        bmi = weight / height ^ 2
        skinny = 18.5
        normal = 25.0
        fat = 30.0 -- ONLY use spaces or tabs when doing these. Otherwise the compiler will get confused. They must ALL be on the same column.

initials :: String -> String -> String
initials firstname lastname = [f] ++ "." ++ [l] ++ "."
        where
                (f:_) = firstname
                (l:_) = lastname

calcBmi :: (RealFloat a) => [(a, a)] -> [a]
calcBmi xs = [bmi w h | (w, h) <- xs]
        where
                bmi weight height = weight / height ^ 2


-- LET --

-- When using let bindings, only the specific portion of the function that you put it in can see it. Since they are expressions, you can put them anywhere you want
-- and not just at the end. The format is "let <binding> in <expression>". Only the <expression> and see what you defined in <binding>
-- Function to find the area of a cylinder
cylinder :: (RealFloat a, Show a) => a -> a -> String
cylinder r h =
        let
                sideArea = 2 * pi * r * h
                topArea = pi * r ^ 2
        in
                "The area of a cylinder with a radius of " ++ show r ++ " and a height of " ++ show h ++ " is " ++ show (sideArea + 2 * topArea)

-- We can use let bindings to make functions as well in a local scope.
square' :: Int -> Int -> Int -> (Int, Int, Int)
square' a b c = let square x = x * x in (square a, square b, square c)

-- If we don't want to bind variables in columns, we can do it in a single line but with semicolons.
noColumns :: (Int, String)
noColumns = (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar="there!" in foo++bar)

-- You can also pattern match with let bindings if you don't want to use semicolons.
noColumns' :: Int
noColumns' = (let (a,b,c) = (1,2,3) in a*b*c) * 100

-- It's also possible to use let bindings inside of list comprehensions.
calcBmis :: (RealFloat a, Show a) => [(a, a)] -> [String]
calcBmis xs = ["Your BMI is " ++ show bmi | (w,h) <- xs, let bmi = w / h ^ 2]

-- We can make it only calculate teh BMI's for fat people.
calcBmis' :: (RealFloat a, Show a) => [(a,a)] -> [String]
calcBmis' xs = ["Your BMI is " ++ show bmi | (w,h) <- xs, let bmi = w/h^2, bmi >= 25.0]


-- CASE --

-- You can use case expressions in place of pattern matching.
head'' :: [a] -> a
head'' xs = case xs of [] -> error "No head on empty lists."
                       (x:_) -> x

-- The pattern for these is pretty simple.
-- case [expression] of [pattern] -> [result]
--                      [pattern] -> [result]
--                      [pattern] -> [result]
--                      ....
-- Where [expression] is matched against the [patterns].

describeList :: [a] -> String
describeList x = "This is a" ++ case x of [] -> "n empty list."
                                          (x:[]) -> " singleton list."
                                          (x:_) -> " longer list."
```