```haskell
--HIGHER ORDER FUNCTIONS--

-- CURRIED FUNCTIONS --

-- ALL Haskell functions only take one parameter. It sounds a little weird, but it kind of makes sense.
-- If you call "max 4 5", it's the same as "(max 4) 5". Doing max 4 5 creates a function that takes a parameter and returns either 4 or that
-- parameter, depending on which is bigger. Then, 5 is applied to that function and it gives us our result.
foo :: (Num a) => a -> a -> a
foo a b = 2 * a - 3 * b
-- If you call "foo 1" it will return a partially applied function. So it will reuturn "foo 1 b = 2 * 1 - 3 * b".
-- let function = foo 1 (returns 'function 1 b = 2*1-3*b')
-- function 3 (reuturns -7 because function 1 3 = -7)
-- You could also say "let multTwoWithNine = multThree 9 in multThree x y z = x*y*z" then call "multTwoWithNine 2 3" and get 54 out of it.
-- Then you could also say "let multWithEighteen = multTwoWithNine 2" so that multWithEighteen x = (((multThree 9) 2) x)

multThree :: (Floating a) => a -> a -> a -> a
multThree x y z = z*y*z
-- Function to compare 100 to a number
compareWithHundred :: (Ord a, Num a) => a -> Ordering
compareWithHundred = compare 100 -- compare 100 returns a function that takes one parameter. Basically "compare 100 x"

-- INFIX FUNCTIONS --
-- Creates a function that is partially applied, but can be used with an operator (*, +, -, /, %)
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)

-- Function to check if a character is upper case
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])

-- Function that subtracts 4. You need to use the "subtract" function with it otherwise it thinks -4 is negative 4.
subtractFour :: (Num a) => a -> a
subtractFour = (subtract 4)

-- Functions that take functions as parameters! It then applies the function to the result of the function.
-- So if you type applyTwice (3*) 10, it does 3*(3*10) => 90
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
-- You can also say 'let functionName = applyTwice (3*)' so that you can type functionName 5 and get teh number 45.

--zipWith function. Takes function as parameter, then returns a zipped list.
-- So calling "zipWith' (*) [1,2,3] [1,2,3]" returns [1,4,9]
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:xy) = f x y : zipWith' f xs xy

-- Simple function to flip the order of the parameters of a function.
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f x y = f y x


-- MAP AND FILTER --
-- These can both be done with list comprehensions, but doing them this way is easier to understand.
-- Map function takes a function and a list and then applies that function to every element in the list.
map' :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs

-- Function to filter out unwanted values from a list.
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' f (x:xs)
        | f x = x : filter' f xs
        | otherwise = filter' f xs

-- Quicksort function redefined using the filter function.
quicksort' :: (Ord a) => [a] -> [a]
quicksort' [] = []
quicksort' (x:xs) =
        let
                smallerSorted = quicksort' (filter' (<=x) xs)
                largerSorted = quicksort' (filter' (x<) xs)
        in smallerSorted ++ [x] ++ largerSorted

-- Finds the largest number that is divisible by 3829.
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
        where p x = x `mod` 3829 == 0

-- takeWhile takes a predicate and a list. It looks over the list and stops taking once it finds a place in the list where the predicate fails.
-- sum (takeWhile (<10000) (filter odd (map (^2) [1..]))) will give us the sum of all odd squares that are less than 10,000.
-- takeWhile (/=' ') "some string goes here" will give us the first word in a string.

-- Collatz sequences
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
        | even n = n:chain (n `div` 2)
        | odd n = n:chain (n*3 + 1)

-- This gets the number of "long chains." AKA the number of chains that are longer than length 15.
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
        where isLong xs = length xs > 15

-- LAMBDAS --
-- These are used basically make anonymous functions. They don't need a name and they are applied with \, which kind of looks like a lambda :P
-- We can re-write the numLongChains function using lambdas instead of the where binding.
numLongChains' :: Int
numLongChains' = length (filter (\xs -> length xs > 15) (map chain [1..100]))
-- Lambdas are just expressions, so we can pass them wherever an expression can be passed.
-- Lambdas can take any number of parameters. They are written in the form of (\PARAM PARAM PARAM -> FUNCTION)
-- You can also pattern match with lambdas, but you need to be careful because if there is an error, it doesn't fall through.
lambdas = map (\(a,b) -> a+b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
-- While pattern matching, you can only have one pattern you're looking for.

addThree :: (Num a) => a -> a -> a -> a
addThree a b c = a + b + c
-- is the same as
addThree' :: (Num a) => a -> a -> a -> a
addThree' = (\a -> (\b -> (\c -> a+b+c)))

-- FOLDL --
-- foldl takes a binary function, a starting position, and a list. It then applies that function to the elements in the list.
-- We can rewrite the sum function like this with foldl:
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\strt x -> strt + x) 0 xs
-- It basically says strt = 0, xs = [1,2,3,4]. It'll preform the anonymous function on the two parameters and then reset the strt value to the value of strt + x
-- where x is just one of the elements in the list that is provided.

-- We can also rewrite the sum function a lot easier.
sum'' :: (Num a) => [a] -> a
sum'' = foldl (+) 0
-- We can replace the lambda (\strt x -> strt + x) because that's the same as (+). We can omit the xs because calling foldl (+) 0 will return a function that takes a list
-- because of currying. So if you have a function foo a = foo b a, you can rewrite that as foo = bar b
five :: (Num a, Eq a) => [a] -> Bool
five = elem 1

-- elem function written with left folds.
elem' :: (Eq a) => a -> [a] -> Bool
elem' a = foldl (\acc x -> if x == a then True else acc) False

-- FOLDR --
-- foldr works exactly the same as foldl but it starts from the right side of the list instead of the left side.
-- map function with foldr
map'' :: (a -> b) -> [a] -> [b]
map'' f a = foldr (\x acc -> f x : acc) [] a

-- Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that.
-- Whenever you want to traverse a list to return something, chances are you want a fold.
-- WHEN BUILDING A NEW LIST FROM A LIST, USE RIGHT FOLD BECAUSE USING : IS LESS RESOURCE-EXPENSIVE THAN ++.
-- Right folds work on infinite lists if you have a starting point. Cause it can start from a point and get to the beginning. But this can't
-- work with left folds because you can't go to infinity without an unending loop.

-- FOLDL1 --
-- foldl1 works just like foldl except it assumes that the starting value is the first element of the list that is provided.
-- So much currying.
sum''' :: (Num a) => [a] -> a
sum''' = foldl1 ( + )

-- FOLDR1 --
-- foldr1 works just like foldr but it uses the last element of the list provided as the starting value

-- COMMMON FUNCTIONS WRITTEN WITH FOLDS --
-- maximum function written with folds.
maximum' :: (Ord a) => [a] -> a
maximum' = foldl1 (\acc x -> if x > acc then x else acc)

-- reverse function written with folds
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

-- product function with folds
product' :: (Num a) => [a] -> a
product' = foldl1 ( * )

-- filter function with folds
filter'' :: (a -> Bool) -> [a] -> [a]
filter'' f = foldr (\x acc -> if f x then x : acc else acc) []

-- head function with folds
head' :: [a] -> a
head' = foldr1 (\x _ -> x)

-- last function with folds
last' :: [a] -> a
last' = foldl1 (\_ x -> x)

-- SCANR --
-- scanr works like foldr, only it just reports the accumulator value each time it goes over a new element of the list.
sumAcc :: (Num a) => [a] -> [a]
sumAcc = scanr ( + ) 0

-- SCANL --
-- scanl works like scanr, but it starts from the left.
sumAcc' :: (Num a) => [a] -> [a]
sumAcc' = scanr ( + ) 0

-- How many numbers does it take for the sum of the squares of those numbers to exceed 1000?
sqrtSums :: Int
sqrtSums = length (takeWhile (< 1000) (scanl ( + ) 1 (map sqrt [1..])))

-- FUNCTION APPLICATION --
-- Using $ has the lowest precedence. It just helps us so that we don't have to write as many parenthesis.
-- So if you wanted to take the squares root of three and add it to 4 and 9, you'd do sqrt 3 + 4 + 9, but if you wanted to
-- add the squares roots of all those numbers you could write sqrt $ 3 + 4 + 9 instead of sqrt (3 + 4 + 9)
-- Example
randomness :: Int
randomness = sum (filter (> 10) (map (*2) [2..10]))
-- This function can be rewritten with function application like this:
randomness' :: Int
randomness' = sum $ filter (>10) $ map (*2) [2..10]

-- FUNCTION COMPOSITION --
-- Using the '.' character, we can do function composition like we can with math.
-- So f(g(x)) = (f . g) x
-- IMPORTANT: f must take a value that is the same type as g's return value.

-- Function that turns all numbers negative.
negate' :: (Num a) => [a] -> [a]
negate' = map (\x -> negate (abs x))
-- This is rather annoying to do and can be a little confusing. We can do the same with function composition much easier.
negate'' :: (Num a) => [a] -> [a]
negate'' = map $ negate . abs

-- More examples of how function composition can be useful.
mathIsFun :: (Num a) => [[a]] -> [a]
mathIsFun = map (\xs -> negate (sum (tail xs)))
-- can be turned into this:
mathIsFun' :: (Num a) => [[a]] -> [a]
mathIsFun' = map $ negate . sum . tail

-- Sum of all odd squares that are less than 10000
sumOddSquares :: Integer
sumOddSquares = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
-- can be turned into this
sumOddSquares' :: Integer
sumOddSquares' = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
-- which can be written more easily like this
sumOddSquares'' :: Integer
sumOddSquares'' =
        let
                oddSquares = filter odd . map (^2) $ [1..]
                belowLimit = takeWhile (<10000) oddSquares
        in sum belowLimit
```