**Academic Year 2024-25 Even**

**19CSE313 – Principles of Programming Language**

**B.Tech CSE 2022-26 F Section**

**Practice Set 5 - Higher Order Function**

1. multThree :: (Num a) => a -> a -> a -> a

multThree x y z = x \* y \* z

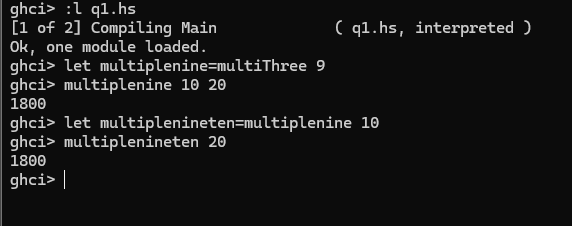
ghci> let multTwoWithNine = multThree 9

ghci> multTwoWithNine 2 3

ghci> let multWithEighteen = multTwoWithNine 2

ghci> multWithEighteen 10

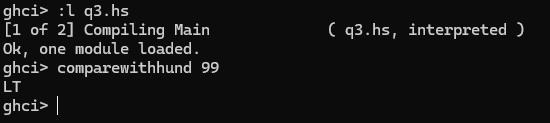
ghci> multThree 3 4



1. compareWithHundred :: (Num a, Ord a) => a -> Ordering

compareWithHundred x = compare 100 x

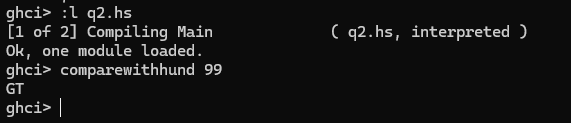
ghci> compareWithHundred 99



1. compareWithHundred :: (Num a, Ord a) => a -> Ordering

compareWithHundred = compare 100

ghci> compareWithHundred 99

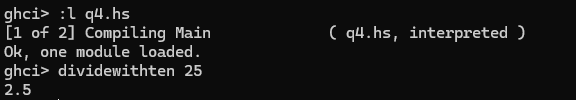


The difference between 2 and 3 question is in q2 the argument is passed into function while in q3 the compare withhund is a defined function and inferred automatically.

1. divideByTen :: (Floating a) => a -> a

divideByTen = (/10)

ghci> divideByTen 200

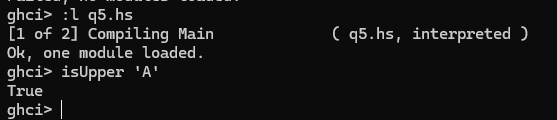


1. isUpperAlphabetic :: Char -> Bool

isUpperAlphabetic = (`elem` ['A'..'Z'])

ghci> isUpperAlphabetic 'A'

ghci> isUpperAlphabetic 'd'



1. subtract4 :: (Num a) => a -> a

subtract4 = (subtract 4)

ghci> subtract4 16

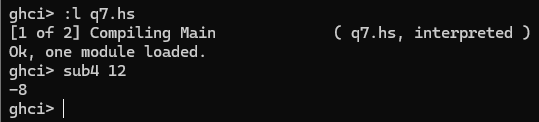


🡪it called as arg-4,

1. subtract4 :: (Num a) => a -> a

subtract4 = ((-) 4)

ghci> subtract4 16



🡪it call as 4-arg .since (-) is taken as inflix not as -4

1. applyTwice :: (a -> a) -> a -> a

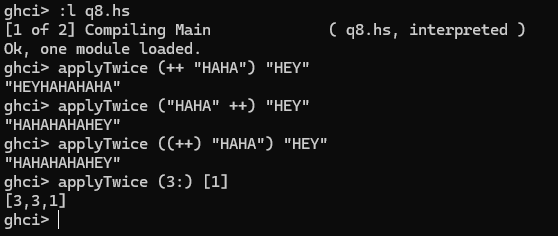
applyTwice f x = f (f x)

ghci> applyTwice (+3) 10

ghci> applyTwice (++ " HAHA") "HEY"

ghci>applyTwice ("HAHA " ++) "HEY"

ghci>applyTwice (3:) [1]



🡪here in case 1 and 2 it is just like concaztenation while in case 3 it takes as influx and for case 4 (3: ) is the function that is going to call twice of list

1. zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith' \_ [] \_ = []

zipWith' \_ \_ [] = []

zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys

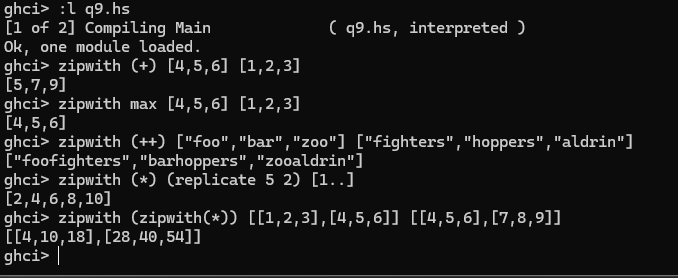
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]

ghci> zipWith' max [6,3,2,1] [7,3,1,5]

ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]

ghci>zipWith' (\*) (replicate 5 2) [1..]

ghci> zipWith' (zipWith' (\*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]



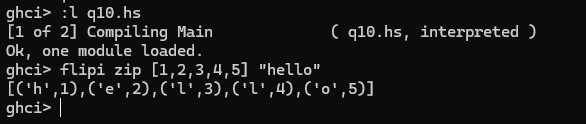
🡪in case 1 and 2,3:it doing addition and max,concatenation operation for element in each l;ist with corresponding element while in case 4 we are creating a list by replicating 2 5 times and doping \* operation ,in case 5 we aree calling f(f(x)),

1. flip' :: (a -> b -> c) -> (b -> a -> c)

flip' f = g

where g x y = f y x

ghci> flip' zip [1,2,3,4,5] "hello"



🡪in this we are doing g(f(x)) for f(g(x))

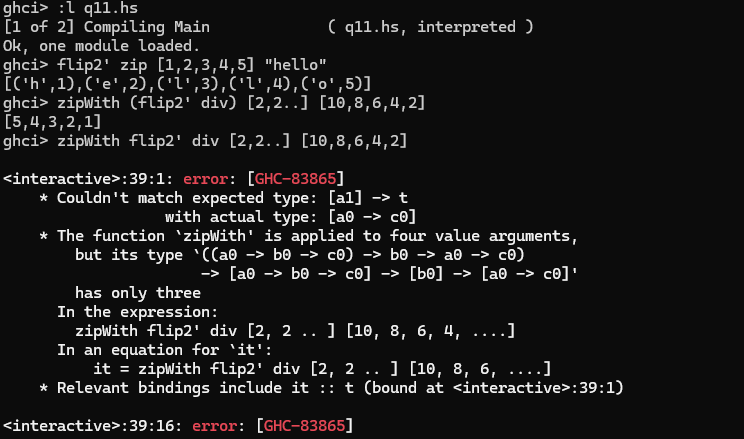
1. flip2' :: (a -> b -> c) -> b -> a -> c

flip2' f y x = f x y

ghci> flip2' zip [1,2,3,4,5] "hello"

ghci> zipWith (flip2' div) [2,2..] [10,8,6,4,2]

ghci> zipWith flip2' div [2,2..] [10,8,6,4,2]



🡪in this zipwith flip2’ div will be the function and eaxch element in 2 list will be passed every time and evaluated,while if there is no () for flip2’ div it is not calculayed as a single function

1. map :: (a -> b) -> [a] -> [b]

map \_ [] = []

map f (x:xs) = f x : map f xs

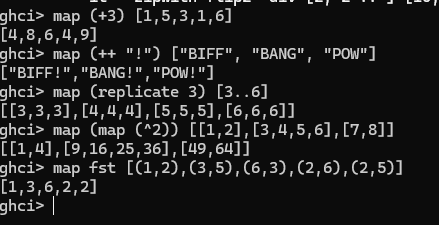
ghci> map (+3) [1,5,3,1,6]

ghci> map (++ "!") ["BIFF", "BANG", "POW"]

ghci> map (replicate 3) [3..6]

ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]

ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]



1. filter :: (a -> Bool) -> [a] -> [a]

filter \_ [] = []

filter p (x:xs)

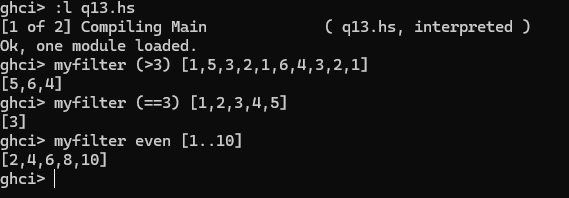
| p x = x : filter p xs

| otherwise = filter p xs

ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]

ghci> filter (==3) [1,2,3,4,5]

ghci> filter even [1..10]



1. quicksort :: (Ord a) => [a] -> [a]

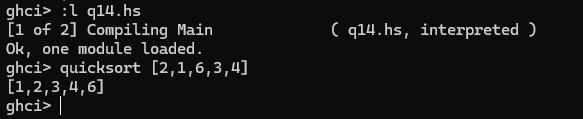
quicksort [] = []

quicksort (x:xs) =

let smallerSorted = quicksort (filter (<=x) xs)

biggerSorted = quicksort (filter (>x) xs)

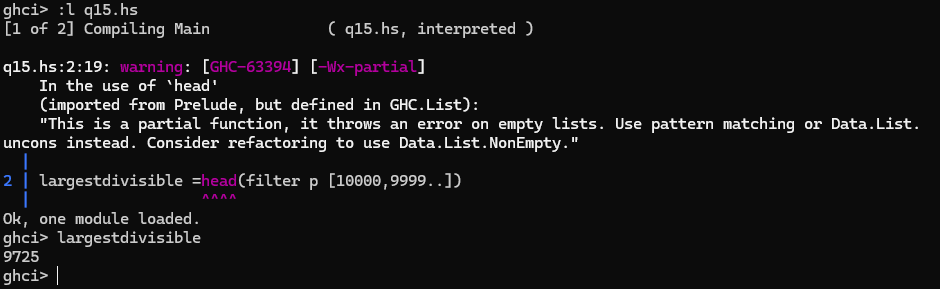
in smallerSorted ++ [x] ++ biggerSorted



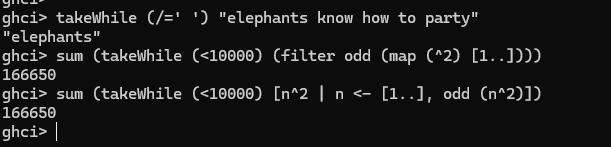
1. largestDivisible :: (Integral a) => a

largestDivisible = head (filter p [100000,99999..])

where p x = x `mod` 3829 == 0



1. ghci> takeWhile (/=' ') "elephants know how to party"
2. ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
3. ghci> sum (takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)])



1. chain :: (Integral a) => a -> [a]

chain 1 = [1]

chain n

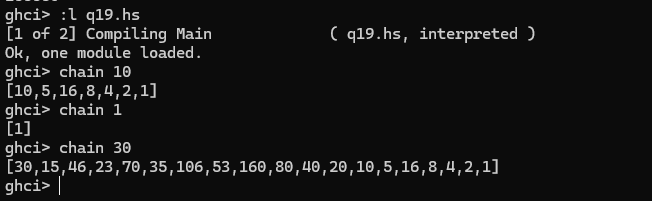
| even n = n:chain (n `div` 2)

| odd n = n:chain (n\*3 + 1)

ghci> chain 10

ghci> chain 1

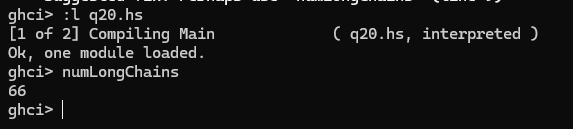
ghci> chain 30



1. numLongChains :: Int

numLongChains = length (filter isLong (map chain [1..100]))

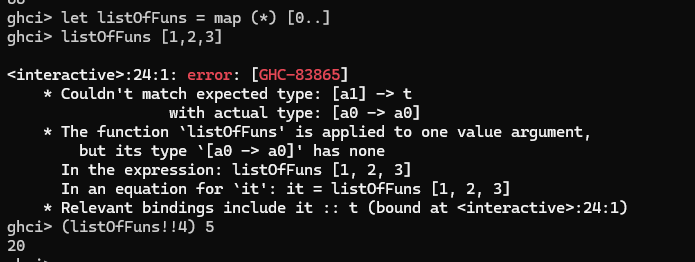
where isLong xs = length xs > 15



**Lambda Functions**

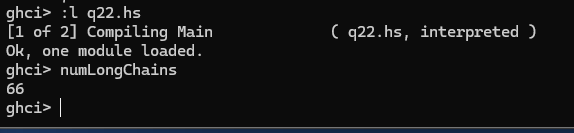
1. ghci> let listOfFuns = map (\*) [0..]

ghci> (listOfFuns !! 4) 5



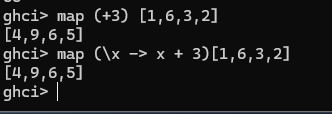
1. numLongChains :: Int

numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))

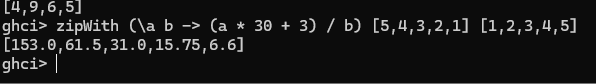


1. ghci> map (+3) [1,6,3,2]

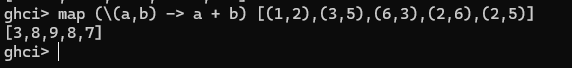
ghci> map (\x -> x + 3)[1,6,3,2]



1. ghci> zipWith (\a b -> (a \* 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]



1. ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]

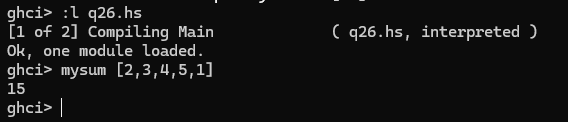


**FOLDL/FOLDR**

1. mysum :: (Num a) => [a] -> a

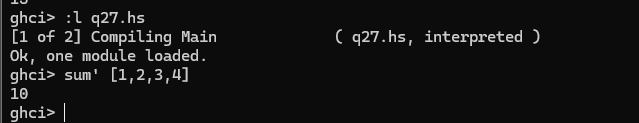
mysum [] = 0

mysum (x:xs) = x + sum xs



1. sum' :: (Num a) => [a] -> a

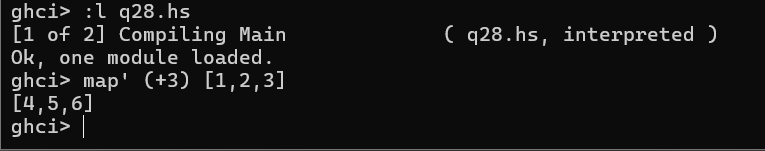
sum' xs = foldl (\acc x -> acc + x) 0 xs



1. map' :: (a -> b) -> [a] -> [b]

map' f xs = foldr (\x acc -> f x : acc) [] xs

ghci> map' (+3) [1,2,3]

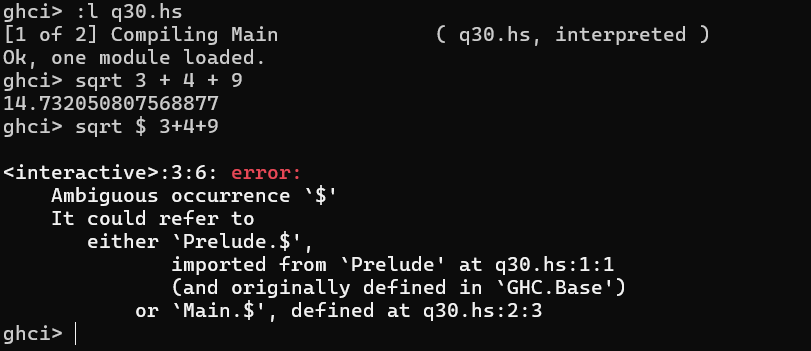


1. Write your observations on the following function implementations using FOLDL and FOLDR
2. ($) :: (a -> b) -> a -> b

f $ x = f x

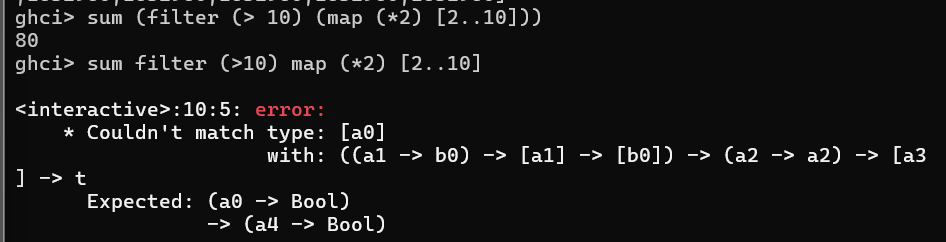
ghci> sqrt 3 + 4 + 9

ghci> sqrt $ 3+4+9

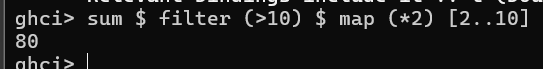


1. ghci> sum (filter (> 10) (map (\*2) [2..10]))

sum filter (>10) map (\*2) [2..10]



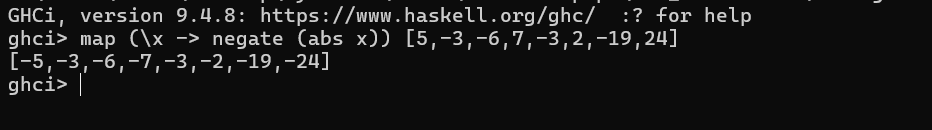
ghci> sum $ filter (>10) $ map (\*2) [2..10]

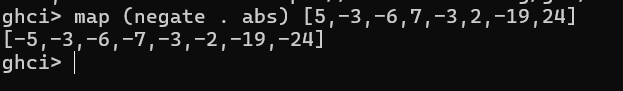


**Composition**

1. ghci>map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]

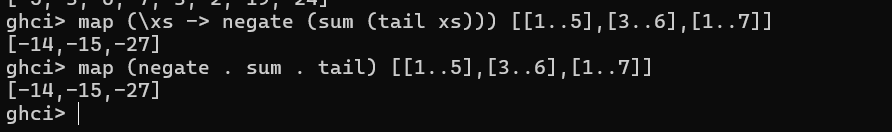
ghci>map (negate . abs) [5,-3,-6,7,-3,2,-19,24]





1. ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]

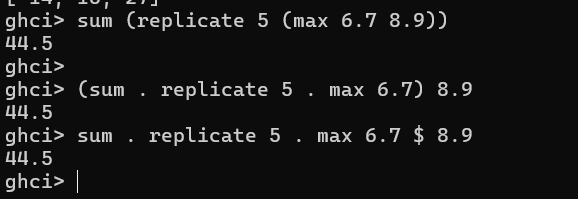
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]



1. ghci> sum (replicate 5 (max 6.7 8.9))

ghci> (sum . replicate 5 . max 6.7) 8.9

ghci> sum . replicate 5 . max 6.7 $ 8.9



1. -- Define two functions

addOne :: Int -> Int

addOne x = x + 1

multiplyByTwo :: Int -> Int

multiplyByTwo x = x \* 2

-- Compose the functions

addOneThenMultiplyByTwo :: Int -> Int

addOneThenMultiplyByTwo = multiplyByTwo . addOne

-- Usage

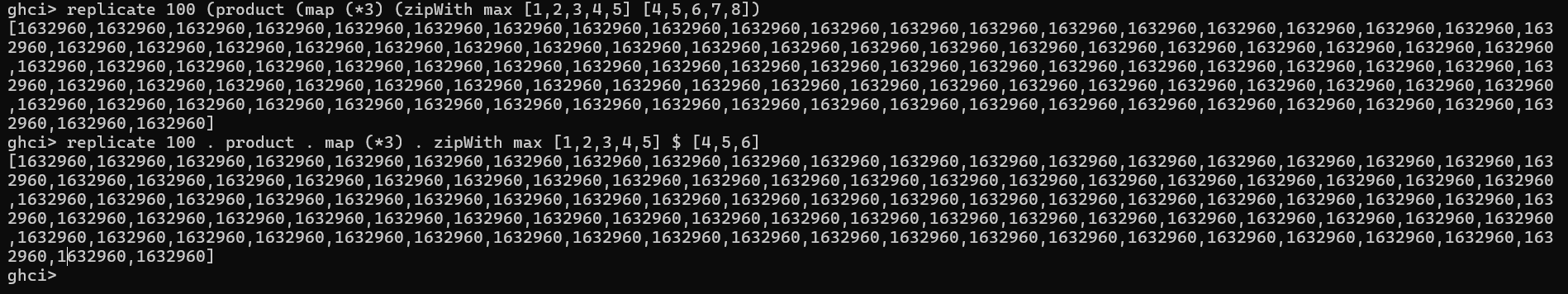
main :: IO ()

main = print (addOneThenMultiplyByTwo 3) -- Output will be 8



1. ghci>replicate 100 (product (map (\*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))

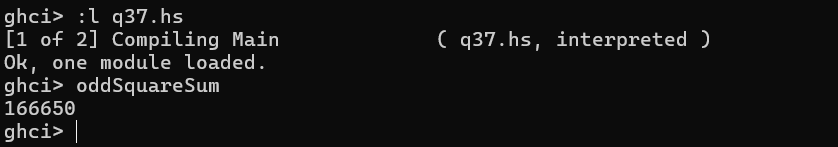
ghci>replicate 100 . product . map (\*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]



1. oddSquareSum :: Integer

oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))

ghci> oddSquareSum



1. oddSquareSum :: Integer

oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]

ghci> oddSquareSum



1. oddSquareSum :: Integer

oddSquareSum =

let oddSquares = filter odd $ map (^2) [1..]

belowLimit = takeWhile (<10000) oddSquares

in sum belowLimit

ghci> oddSquareSum

