**Academic Year 2024-25 Even**

**19CSE313 – Principles of Programming Language**

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

**Practice Set 3 - Haskell List**

1. 1:[]
2. 1:2:3:[]
3. 1:2:[3,4,5]
4. take 10 [2,4..]
5. [x|x<-[2,4..10]]
6. [x\*2 | x <- [1..10]]
7. [x\*x | x <- [1..5]]
8. sqr x = x\*x

[sqr x|x<-[1..5]]

1. [(i,j) | i <- [1..5], j <- [i..5]]
2. [(i,j) | i <- [1..5], even i, j <- [i..5]]
3. [(i,j) | i <- [1..5], even i, j <- [i..5],odd j]
4. [x\*2 | x <- [1..10], x\*2 >= 12]
5. [ x | x <- [50..100], x `mod` 7 == 3]
6. [ x | x <- [50..100], (mod x 7) == 3]



1. boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]

boomBangs [7..13]

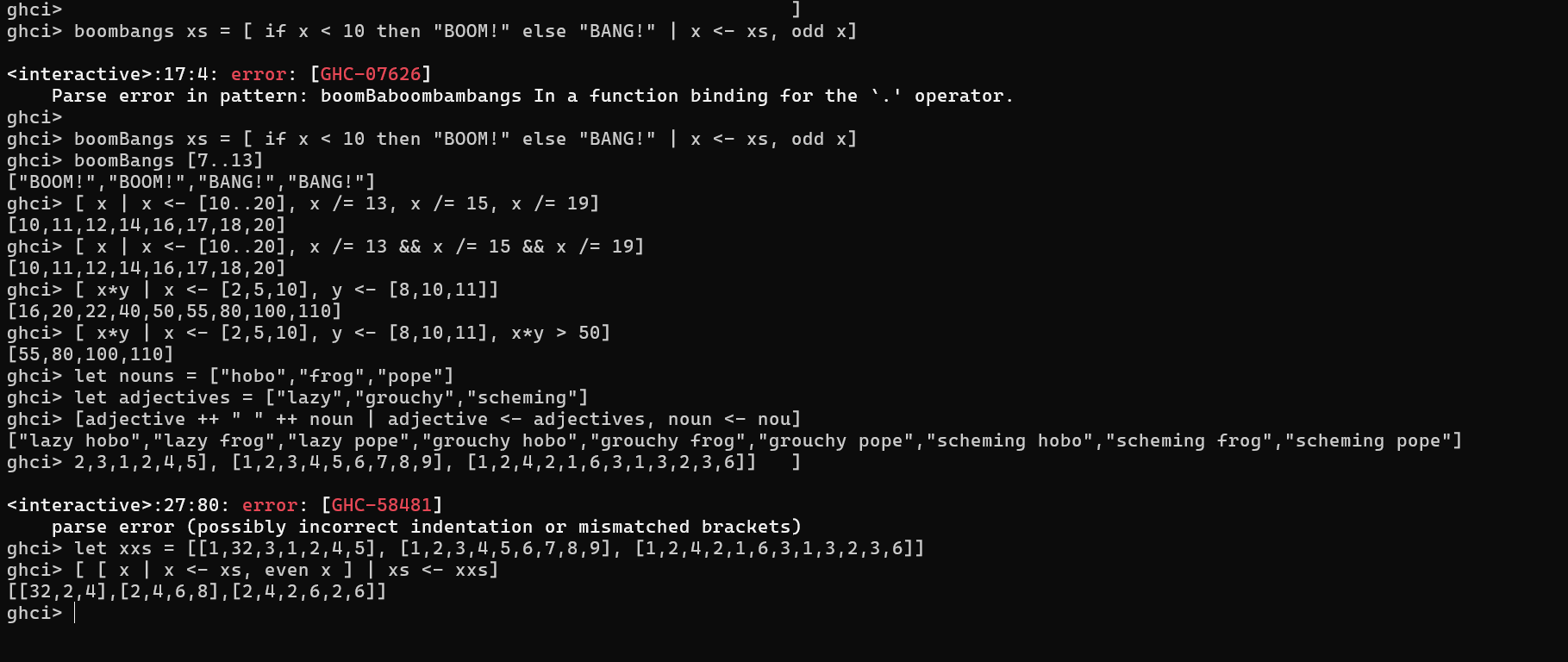
1. [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
2. [ x | x <- [10..20], x /= 13 && x /= 15 && x /= 19]
3. [ x\*y | x <- [2,5,10], y <- [8,10,11]]
4. [ x\*y | x <- [2,5,10], y <- [8,10,11], x\*y > 50]
5. let nouns = ["hobo","frog","pope"]

let adjectives = ["lazy","grouchy","scheming"]

[adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]

1. let xxs = [[1,3,5,2,3,1,2,4,5], [1,2,3,4,5,6,7,8,9], [1,2,4,2,1,6,3,1,3,2,3,6]]

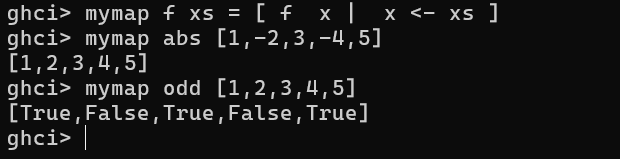
[ [ x | x <- xs, even x ] | xs <- xxs]



1. mymap f xs = [ f x | x <- xs ]

mymap abs [1,-2,3,-4,5]

mymap odd [1,2,3,4,5]



1. myfilter p xs = [x | x <- xs, p x]

myfilter odd [1,2,3,4,5]

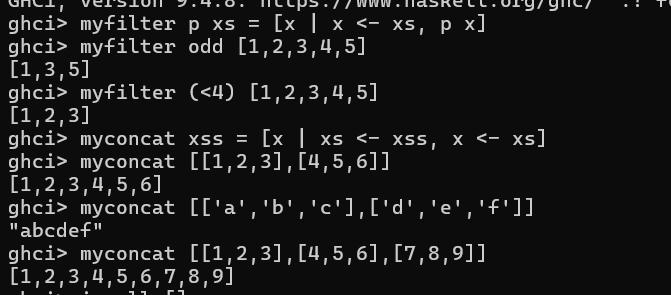
myfilter (<4) [1,2,3,4,5]

1. myconcat xss = [x | xs <- xss, x <- xs]

myconcat [[1,2,3],[4,5,6]]

myconcat [['a','b','c'],['d','e','f']]

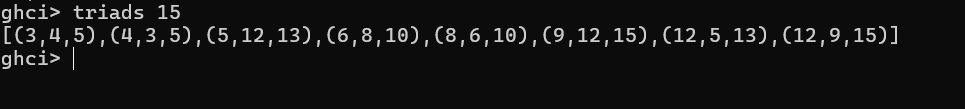
myconcat [[1,2,3],[4,5,6],[7,8,9]]



1. triads :: Int -> [(Int,Int,Int)]

triads n = [(x,y,z) | x <- [1..n], y <- [1..n],z <- [1..n], x\*x+y\*y==z\*z]

triads 15



1. Basic operations
   1. null **(null :: [a] -> Bool)**
      1. null []
      2. null [1,2,3,4,5]
   2. head **(head :: [a] -> a)**
   3. tail **(tail :: [a] -> [a])** 
      1. tail (x:xs) = xs

tail [1,2,3,4,5]

* 1. last **(last :: [a] -> a)**
     1. last [x] = x
     2. last (x:y:ys) = last (y:ys)

last [1,2,3,4,5]

* 1. ++ **((++) :: [a] -> [a] -> [a])**
     1. [] ++ ys = ys
     2. (x:xs) ++ ys = x:(xs ++ ys)

[1,2,3,4,5] ++ [6,7,8,9,10]

* 1. concat1 :: [[a]] -> [a]

concat1 [] = []

concat1 (xs:xss) = xs ++ concat1 xss

concat1 []

concat1 [[1,2,3,4], [5,6,7,8]]

concat1 [[1,2,3,4], [5,6,7,8],[9,10,11,12]]

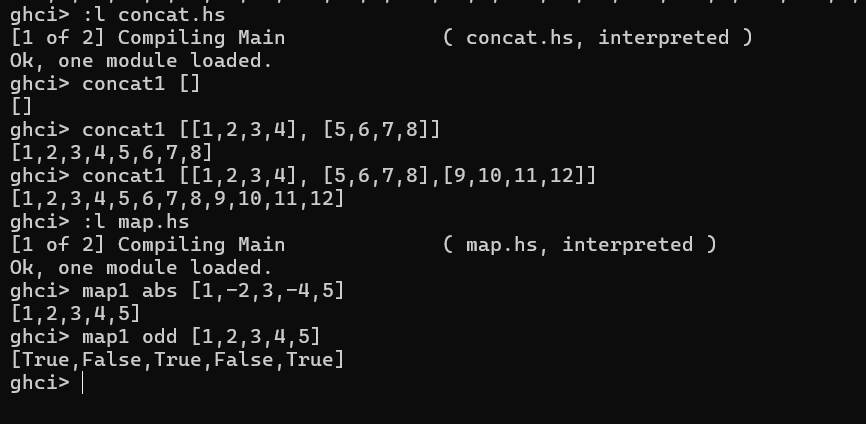
* 1. map1 :: (a -> b) -> [a] -> [b]

map1 f [] = []

map1 f (x:xs) = f x:map1 f xs

map1 abs [1,-2,3,-4,5]

map1 odd [1,2,3,4,5]



* 1. filter1 :: (a -> Bool) -> [a] -> [a]

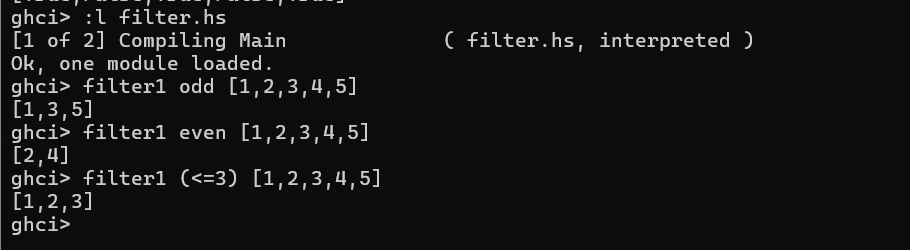
filter1 p [] = []

filter1 p (x:xs) = if p x then x:filter1 p xs else filter1 p xs

filter1 odd [1,2,3,4,5]

filter1 even [1,2,3,4,5]

filter1 (<=3) [1,2,3,4,5]



* 1. :type zip

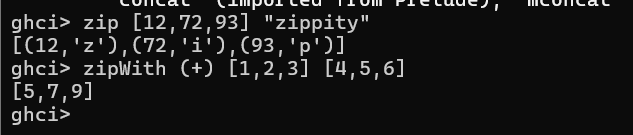
zip :: [a] -> [b] -> [(a, b)]

zip [12,72,93] "zippity"

* 1. :type zipWith

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (+) [1,2,3] [4,5,6]



* 1. zip :: [a] -> [b] -> [(a,b)]

zip (x:xs) (y:ys) = (x,y): zip xs ys

zip \_ \_ = []

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

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

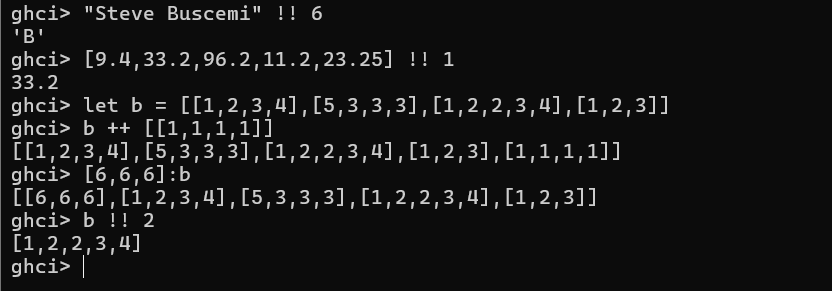
zipWith f \_ \_ = []

* 1. !!
     1. "Steve Buscemi" !! 6
     2. [9.4,33.2,96.2,11.2,23.25] !! 1
     3. let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]

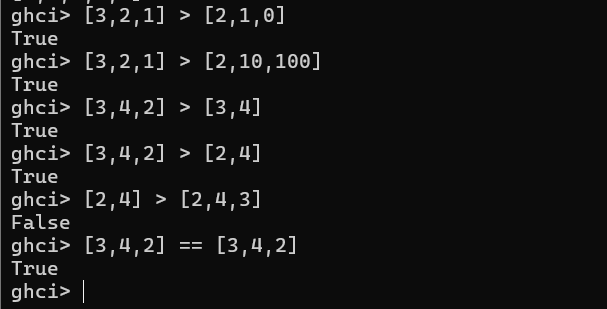
b ++ [[1,1,1,1]]

[6,6,6]:b

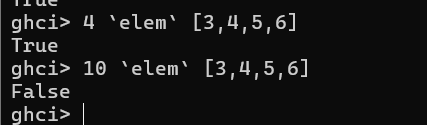
b !! 2



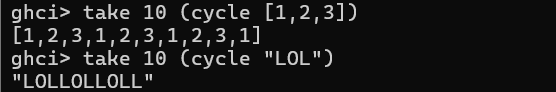
* 1. List Comparison
     1. [3,2,1] > [2,1,0]
     2. [3,2,1] > [2,10,100]
     3. [3,4,2] > [3,4]
     4. [3,4,2] > [2,4]
     5. [2,4] > [2,4,3]
     6. [3,4,2] == [3,4,2]



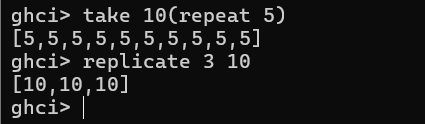
* 1. init
  2. reverse
  3. minimum
  4. maximum
  5. sum
  6. product
  7. elem
     1. 4 `elem` [3,4,5,6]
     2. 10 `elem` [3,4,5,6]



* 1. Cycle
     1. take 10 (cycle [1,2,3])
     2. take 12 (cycle "LOL ")



* 1. repeat
     1. take 10 (repeat 5)
  2. replicate
     1. replicate 3 10

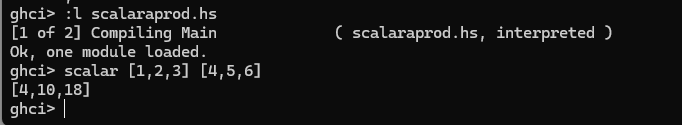


**Write Haskell Programs to do the following tasks using the concepts of Lists and Functions**

1. To find the scalar product of two lists (sum of the product of the corresponding components)

scalar :: [Int] -> [Int] -> [Int]

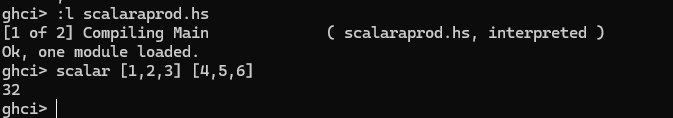
scalar a b = [x \* y | (x, y) <- zip a b]





scalar :: [Int] -> [Int] -> Int

scalar xs ys = sum(zipWith (\*) xs ys)

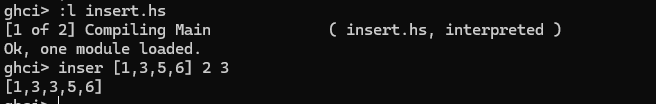


1. Insert a new value in a given position and return the resulting new list

(using built-in func)

inser::[Int]->Int->Int->[Int] –list,pos,ele, returns updated list

inser a b c = take (b-1) a ++ ([c] ++ drop (b-1) a)



(without built-in)

Using recursion:

insertAt :: a -> Int -> [a] -> [a]

insertAt x 0 xs = x : xs -- Insert at the beginning

insertAt x pos [] = [x] -- Insert at the end if list is empty

insertAt x pos (y:ys) = y : insertAt x (pos - 1) ys

1. Delete a value from the prescribed position

del :: [Int]->Int->[Int]

del a b= (take (b-1) a) ++ (drop b a)



(without using built-in)

deleteAt :: Int -> [a] -> [a]

deleteAt \_ [] = [] -- If the list is empty, return an empty list

deleteAt 0 (\_:xs) = xs -- If position is 0, skip the first element

deleteAt pos (x:xs) = x : deleteAt (pos - 1) xs -- Recursively process the list

1. To count the number of elements in the given list

countele :: [Int]->Int

countele []=0

countele (x:xs)=1+countele xs

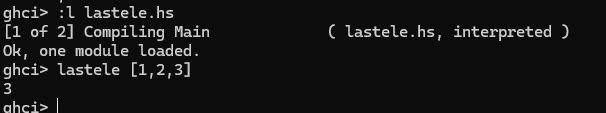


1. Function to print the last element of the list (not to use the last command)

lastele::[Int]->Int

lastele [a]=a

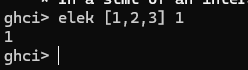
lastele (x:y:ys)=lastele(y:ys)



1. To print the Kth element in the list

elek :: [Int] -> Int -> Int

elek a b = head (drop (b - 1) a)



kthElement :: Int -> [a] -> a

kthElement 0 (x:\_) = x -- Return the head if position is 0

kthElement k (\_:xs) = kthElement (k - 1) xs -- Recur with the tail and decrement position

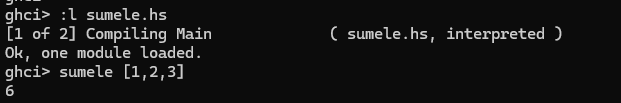
kthElement \_ [] = error "Index out of bounds" -- Error if the position is out of bounds

7) sum of the elements

sumele :: [Int]->Int

sumele []=0

sumele (x:xs)= x+sumele xs



8)

Maximum element

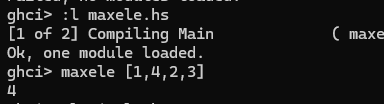
maxele::[Int]->Int

maxele(x:xs)=maxhelp xs x

maxhelp ::[Int]->Int->Int

maxhelp [] currmax=currmax

maxhelp (y:ys) currmax=maxhelp ys (if y>currmax then y else currmax)



Minelement:

minele::[Int]->Int

minele(x:xs)=minhelp xs x

minhelp ::[Int]->Int->Int

minhelp [] currmin=currmin

minhelp (y:ys) currmin=minhelp ys (if y<currmin then y else currmin)

