**Academic Year 2024-25 Even**

**19CSE313 – Principles of Programming Language**

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

**Practice Set 4 - Pattern Matching and Recursion**

1. sayMe :: (Integral a) => a -> String

sayMe 1 = "One!"

sayMe 2 = "Two!"

sayMe 3 = "Three!"

sayMe 4 = "Four!"

sayMe 5 = "Five!"

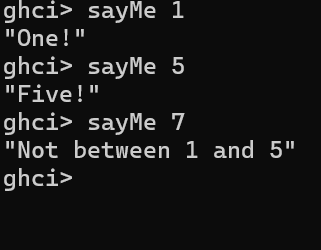
sayMe x = "Not between 1 and 5“

ghci> :l sayMe.hs

ghci> sayMe 1

ghci> sayMe 5

ghci> sayMe 7



1. fact :: Int -> Int

fact 0 = 1

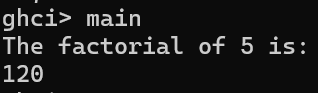
fact n = n \* fact ( n - 1 )

main = do

putStrLn "The factorial of 5 is:"

print (fact 5)

ghci> main



1. factorialp :: (Integral a) => a -> a

factorialp 0 = 1

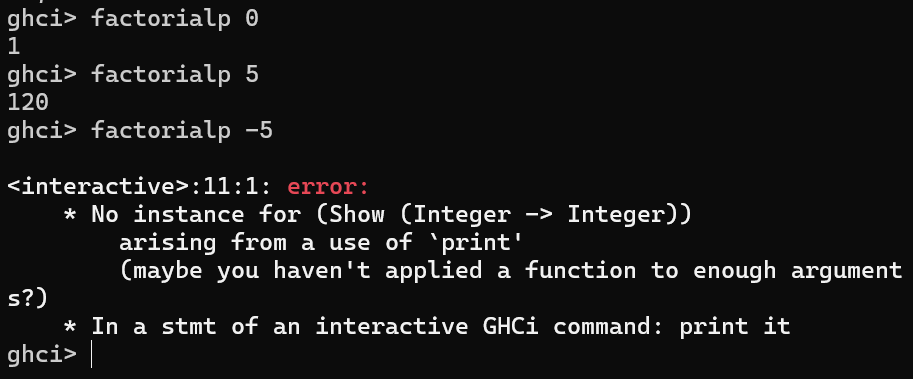
factorialp n = n \* factorialp (n - 1)

ghci> :l factorialp.hs

ghci> factorialp 0

ghci> factorialp 5

ghci> factorialp -5



1. --charName.hs

charName :: Char -> String

charName 'a' = "Albert"

charName 'b' = "Broseph"

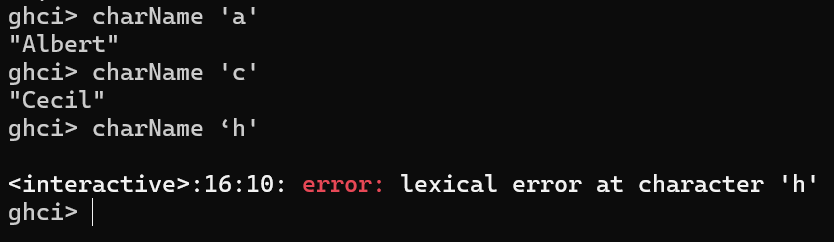
charName 'c' = "Cecil“

ghci>:l charName.hs

ghci> charName 'a'

ghci> charName 'c'

ghci> charName ‘h'

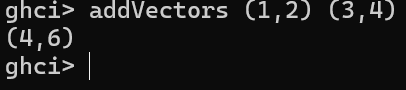


1. Add charName1 x = "String not defined” to the definition of charName and try again
2. addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)

addVectors a b = (fst a + fst b, snd a + snd b)

ghci> :l addVectors.hs

ghci> addVectors (1,2) (3,4)



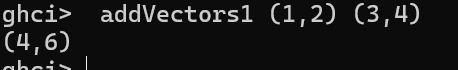
1. --addVectors1.hs - with pattern matching

addVectors1 :: (Num a) => (a, a) -> (a, a) -> (a, a)

addVectors1 (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

ghci> :l addVectors1.hs

ghci> addVectors1 (1,2) (3,4)



1. - - tripletFunctions.hs

first :: (a, b, c) -> a

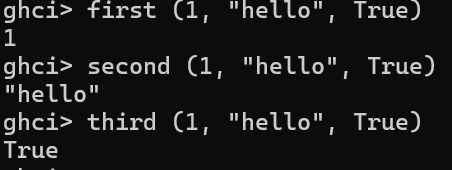
first (x, \_, \_) = x

second :: (a, b, c) -> b

second (\_, y, \_) = y

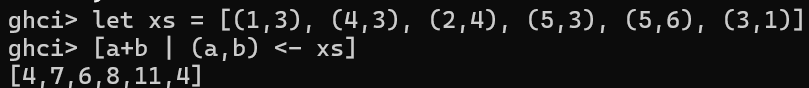
third :: (a, b, c) -> c

third (\_, \_, z) = z



1. let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]

[a+b | (a,b) <- xs]



1. --myhead.hs

myhead :: [a] -> a

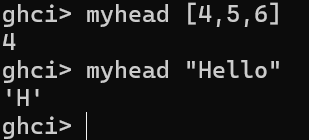
myhead [] = error "Can't call head on an empty list, dummy!"

myhead (x:\_) = x

ghci> :l myhead.hs

ghci> myhead [4,5,6]

ghci> myhead "Hello"



1. --tell.hs

tell :: (Show a) => [a] -> String

tell [] = "The list is empty"

tell (x:[]) = "The list has one element: " ++ show x

tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y

tell (x:y:\_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y

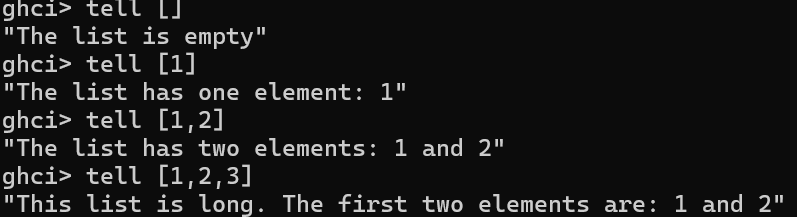
ghci> :l tell.hs

ghci> tell []

ghci> tell [1]

ghci> tell [1,2]

ghci> tell [1,2,3]



1. --mylength.hs

mylength :: (Num b) => [a] -> b

mylength [] = 0 -- length of empty list

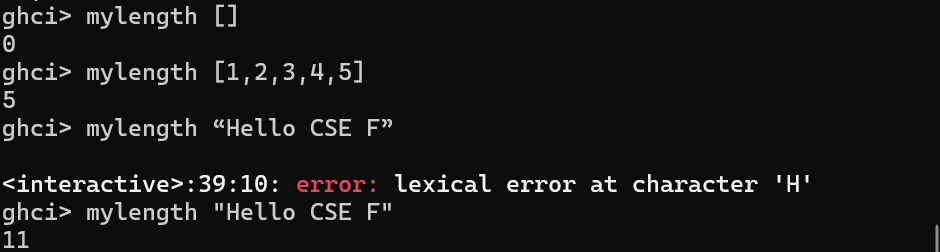
mylength (\_:xs) = 1 + mylength xs -- recursive call to mylength

ghci> :l mylength.hs

ghci> mylength []

ghci> mylength [1,2,3,4,5]

ghci> mylength “Hello CSE F”



1. --mysum.hs

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

mysum [] = 0

mysum (x:xs) = x + mysum xs

ghci> :l mysum.hs

ghci> mysum [1,2,3,4,5]



1. --bmiTell.hs

bmiTell :: (RealFloat a) => a -> String

bmiTell bmi

| bmi <= 18.5 = "You're underweight"

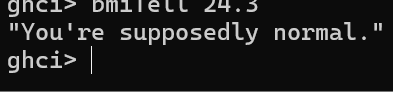
| bmi <= 25.0 = "You're supposedly normal.”

| bmi <= 30.0 = "You’re overweight”

| otherwise = "You’re obese”

ghci> :l bmiTell.hs

ghci> bmiTell 24.3



1. --bmiTell1.hs

bmiTell1 :: (RealFloat a) => a -> a -> String

bmiTell1 weight height

| weight / height ^ 2 <= 18.5 = "You're underweight"

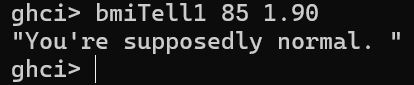
| weight / height ^ 2 <= 25.0 = "You're supposedly normal. "

| weight / height ^ 2 <= 30.0 = "You’re overweight!"

| otherwise = "You’re obese!”

ghci> :l bmiTell.hs

ghci> bmiTell1 85 1.90



1. --mymax.hs

mymax :: (Ord a) => a -> a -> a

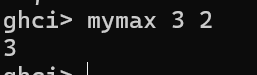
mymax a b

| a > b = a

| otherwise = b

ghci> :l mymax.hs

ghci> mymax 3 2



1. myCompare :: (Ord a) => a -> a -> Ordering

a `myCompare` b

| a > b = GT

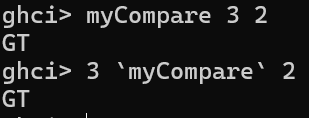
| a == b = EQ

| otherwise = LT

ghci> :l mymax.hs

ghci> myCompare 3 2

ghci> 3 `myCompare` 2



1. roots :: (Float, Float, Float) -> (Float, Float)

roots (a,b,c) = (x1, x2) where

x1 = e + sqrt d / (2 \* a)

x2 = e - sqrt d / (2 \* a)

d = b \* b - 4 \* a \* c

e = - b / (2 \* a)

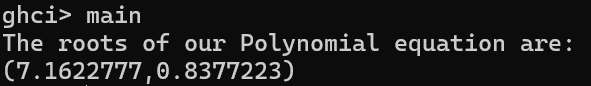
main = do

putStrLn "The roots of our Polynomial equation are:"

print (roots(1,-8,6))

ghci> :l whereroots.hs

ghci> main



1. **Modify bmiTell1.hs using ‘where’ clause**

bmiTell :: (RealFloat a) => a -> a -> String

bmiTell weight height

| bmi <= 18.5 = "You're underweight"

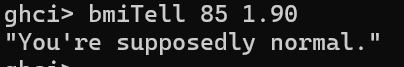
| bmi <= 25.0 = "You're supposedly normal."

| bmi <= 30.0 = "You’re overweight"

| otherwise = "You’re obese"

where

bmi = weight / height^2



1. bmiTell :: (RealFloat a) => a -> a -> String

bmiTell weight height

| bmi <= skinny = "You're underweight!"

| bmi <= normal = "You're supposedly normal."

| bmi <= fat = "You're overweight"

| otherwise = "You're obese"

where bmi = weight / height ^ 2

skinny = 18.5

normal = 25.0

fat = 30.0



1. bmiTell :: (RealFloat a) => a -> a -> String

bmiTell weight height

| bmi <= skinny = "You're underweight!"

| bmi <= normal = "You're supposedly normal."

| bmi <= fat = "You're overweight"

| otherwise = "You're obese"

where bmi = weight / height ^ 2

(skinny, normal, fat) = (18.5, 25.0, 30.0)



1. –createInitials.hs

initials :: String -> String -> String

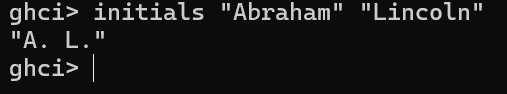
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."

where (f:\_) = firstname

(l:\_) = lastname

ghci>:l createInitials.hs

ghci>initials "Abraham" "Lincoln"



1. - - surfarea.hs

cylinder :: (RealFloat a) => a -> a -> a

cylinder r h =

let sideArea = 2 \* pi \* r \* h

topArea = pi \* r ^2

in sideArea + 2 \* topArea

ghci>:l surfarea.hs

ghci>cylinder 20 30



1. ghci> 4 \* (if 10 > 5 then 10 else 0) + 2



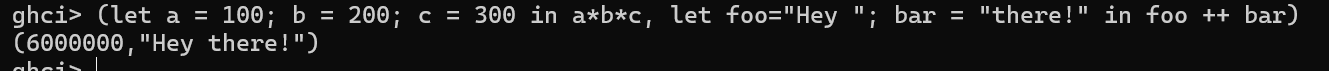
1. ghci> 4 \* (let a = 9 in a + 1) + 2



1. [let square x = x \* x in (square 5, square 3, square 2)]



1. (let a = 100; b = 200; c = 300 in a\*b\*c, let foo="Hey "; bar = "there!" in foo ++ bar)

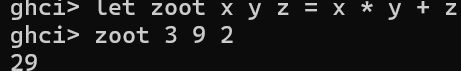


1. (let (a,b,c) = (1,2,3) in a+b+c) \* 100



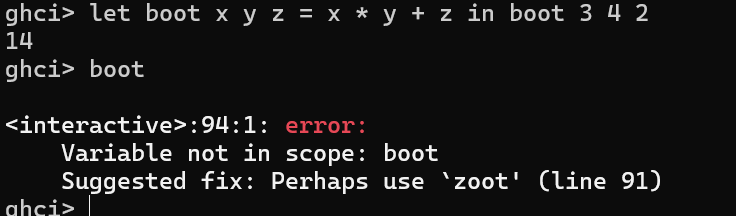
1. ghci> let zoot x y z = x \* y + z

ghci> zoot 3 9 2



1. ghci> let boot x y z = x \* y + z in boot 3 4 2

ghci> boot

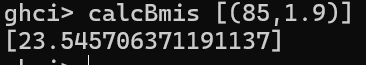


1. calcBmis :: (RealFloat a) => [(a, a)] -> [a]

calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]

ghci>:l bmi.hs

ghci> calcBmis [(85,1.9)]



1. describeList :: [a] -> String

describeList xs = "The list is " ++ case xs of [] -> "empty."

[x] -> "a singleton list."

xs -> "a longer list."

1. describeList :: [a] -> String

describeList xs = "The list is " ++ what xs where

what [] = "empty."

what [x] = "a singleton list."

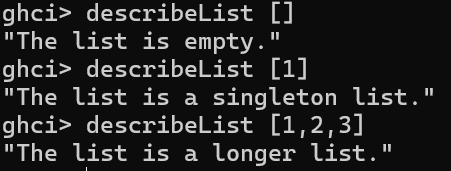
what xs = "a longer list."

ghci>:l listDescription.hs

ghci> describeList []

ghci> describeList [1]

ghci>describeList [1,2,3]



**Recursion**

--maxrec.hs

maxrec :: (Ord a) => [a] -> a

maxrec [] = error "maximum of empty list"

maxrec [x] = x

maxrec (x:xs)

| x > maxTail = x

| otherwise = maxTail

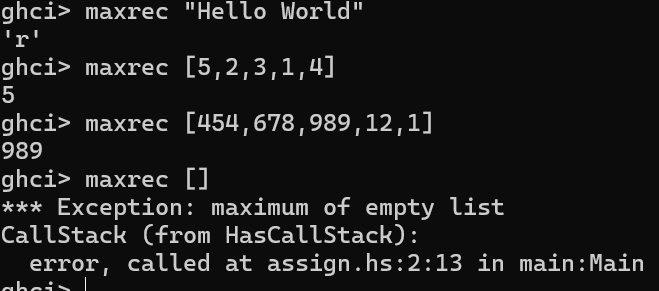
where maxTail = maxrec xs

ghci> maxrec "Hello World"

ghci> maxrec [5,2,3,1,4]

ghci> maxrec [454,678,989,12,1]

ghci> maxrec []



----------------------------------------------------------------------------------------------------------------

--maxrec1.hs

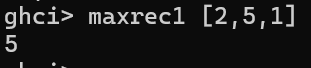
maxrec1 :: (Ord a) => [a] -> a

maxrec1 [] = error "maximum of empty list"

maxrec1 [x] = x

maxrec1 (x:xs) = max x (maxrec1 xs)

ghci> maxrec1 [2,5,1]



---------------------------------------------------------------------------------------------------------------

--myreplicate.hs

myreplicate :: (Num i, Ord i) => i -> a -> [a]

myreplicate n x

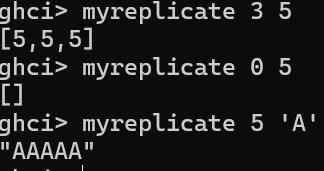
| n <= 0 = []

| otherwise = x:myreplicate (n-1) x

ghci> myreplicate 3 5

ghci> myreplicate 0 5

ghci> myreplicate 5 'A'



------------------------------------------------------------------------------------------------------------

--mytake.hs

mytake :: (Num i, Ord i) => i -> [a] -> [a]

mytake n \_

| n <= 0 = []

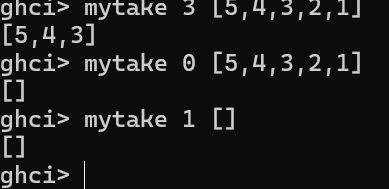
mytake \_ [] = []

mytake n (x:xs) = x : mytake (n-1) xs

ghci> mytake 3 [5,4,3,2,1]

ghci> mytake 0 [5,4,3,2,1]

ghci> mytake 1 []



--------------------------------------------------------------------------------------------------

--myreverse.hs

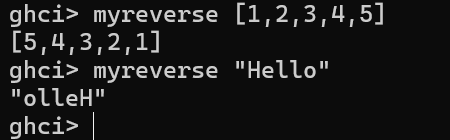
myreverse :: [a] -> [a]

myreverse [] = []

myreverse (x:xs) = myreverse xs ++ [x]

ghci> myreverse [1,2,3,4,5]

ghci>myreverse “Hello”



----------------------------------------------------------------------------------------------

--elem1.hs

elem1 :: (Eq a) => a -> [a] -> Bool

elem1 a [] = False

elem1 a (x:xs)

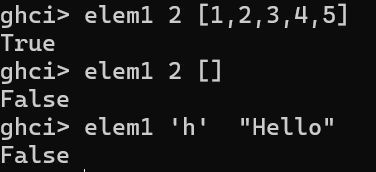
| a == x = True

| otherwise = a `elem1` xs

ghci> elem1 2 [1,2,3,4,5]

ghci> elem1 2 []

ghci> elem1 'h' "Hello"



---------------------------------------------------------------------------------------------

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

quicksort [] = []

quicksort (x:xs) =

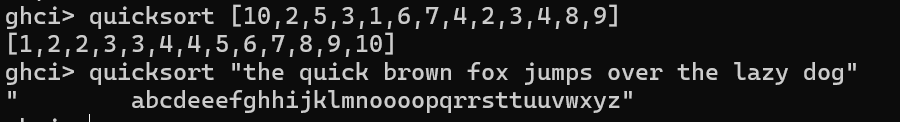
let smallerSorted = quicksort [a | a <- xs, a <= x]

biggerSorted = quicksort [a | a <- xs, a > x]

in smallerSorted ++ [x] ++ biggerSorted

ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]

ghci> quicksort "the quick brown fox jumps over the lazy dog"



--------------------------------------------------------------------------------------------------