CS331: Assignment #4

Due on Monday, February 07, 2022

Siddharth Charan 190101085

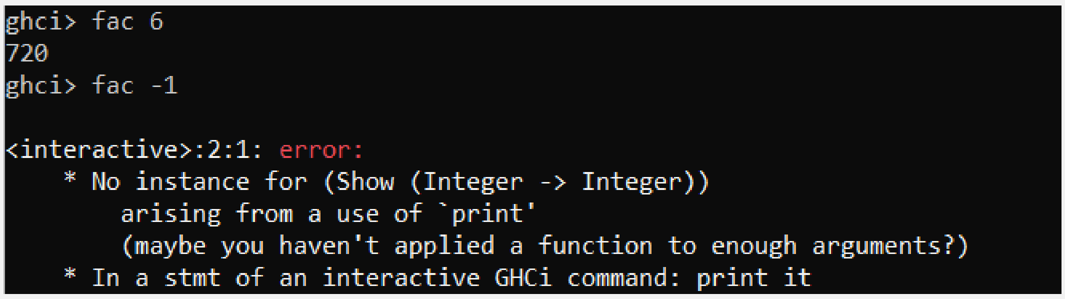
Assignment: Introduction

In this assignment we were asked to solve exercises from 3 chapters 6,7 and 8. 6 and 8 Each had 9 questions in it, 7 had 10 questions, but some questions were asked to leave. These are discussed below and codes are given in separate file.

Question 6.1 : How does the recursive version of the factorial function behave if applied to a negative argument, such as (-1)?

The function does not terminate, because each application of fac decreases the argument by one, and loop will be going towards more negative numbers and hence the base case is never reached, which is fac 0.

We can modify the function like below so that, negative input is prohibited. Code is also given inseparate file.
fac 0 = 1
fac n | n > 0 = n * fac (n-1)



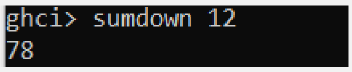
```
ghci> fac 6
720
ghci> fac -1

<interactive>:2:1: error:
    * No instance for (Show (Integer -> Integer))
        arising from a use of `print'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
```

You can note that it is not alllowing -ve numbers input from screenshot.

Question 6.2 : Define a recursive function sumdown :: Int -> Int that returns the sum of the non-negative integers from a given value down to zero.

We can recursively define it like below, code is also given in separate file:

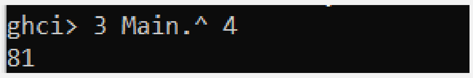
sumdown 0 = 0
sumdown n = n + sumdown (n-1)



```
ghci> sumdown 12
78
```

Question 6.3 : Define the exponentiation operator $\wedge$ for non-negative integers using the same pattern of recursion as the multiplication operator *, and show how the expression $2 \wedge 3$ is evaluated using your definition.

We can define it like below:
(∧) :: Int -> Int -> Int
m ∧ 0 = 1
m ∧ n = m * (m ∧ (n-1))

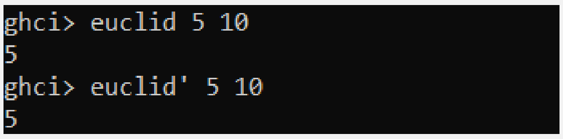


```
ghci> 3 Main.^ 4
81
```

For example, we can have:
2 ∧ 3
= applying ∧
2 * (2 ∧ 2)
= applying ∧
2 * (2 * (2 ∧ 1))
= applying ∧
2 * (2 * (2 * (2 ∧ 0)))
= applying ∧
2 * (2 * (2 * 1))
= applying *
8

Question 6.4 : Define a recursive function euclid :: Int -> Int -> Int that implements Euclid's algorithm for calculating the greatest common divisor of two non-negative integers: if the two numbers are equal, this number is the result; otherwise, the smaller number is subtracted from the larger, and the same process is then repeated

```
euclid x y | x==y = x
           | x < y = euclid x (y-x)
           | y < x = euclid (x-y) y
```



```
ghci> euclid 5 10
5
ghci> euclid' 5 10
5
```

Question 6.5 : Using the recursive definitions given in this chapter, show how length [1,2,3], drop 3 [1,2,3,4,5], and init [1,2,3] are evaluated.

length [1,2,3]
= {applying length}
1 + length [2,3]
= {applying length}
1 + (1 + length [3])
= {applying length}

```
1 + (1 + (1 + length []))
= {applying length}
1 + (1 + (1 + 0))
= {applying +}
1 + (1 + (1))
= {applying +}
1 + (2)
= {applying +}
3

drop 3 [1,2,3,4,5]
= {applying drop}
drop 2 [2,3,4,5]
= {applying drop}
drop 1 [3,4,5]
= {applying drop}
drop 0 [4,5]
= {applying drop}
   [4,5]

init [1,2,3]
= {applying init}
1 : init [2,3]
= {applying init}
1 : 2 : init [3]
= {applying init}
1 : 2 : []
= {list notation}
   [1,2]
```

Question 6.6 : Without looking at the definitions from the standard prelude, define the following library functions on lists using recursion.

We are using ' after each name as, these functions are already defined in prelude.

(a) Decide if all logical values in a list are True: and :: [Bool] -> Bool

```
and' :: [Bool] -> Bool
and' [] = True
and' (x:xs) = x  (and' xs)
```

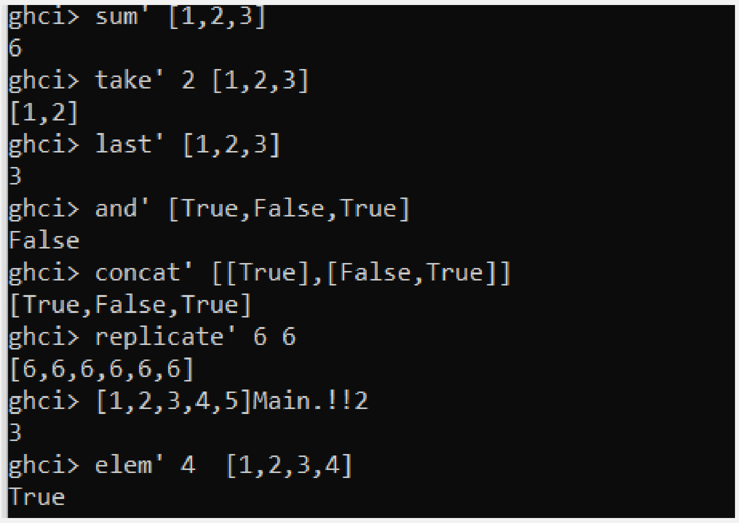
(b) Concatenate a list of lists: concat :: [[a]] -> [a]

```
concat' :: [[a]] -> [a]
concat' [] = []
concat' [x] = x
concat' (x:xs) = x ++ concat' xs
```

(c) Produce a list with n identical elements: replicate :: Int -> a -> [a]

```
replicate' :: Int -> a -> [a]
replicate' 0 _ = []
```

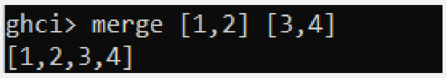
```
replicate’ n x = x : replicate’ (n - 1) x
```
(d) Select the nth element of a list:(!!) :: [a] -> Int -> a
```
(!!) :: [a] -> Int -> a
(!!) (x:_) 0 = x
(!!) (_:xs) n = (Main.!!) xs (n - 1)
```
(e) Decide if a value is an element of a list: elem :: Eq a => a -> [a] -> Bool
```
elem’ :: Eq a => a -> [a] -> Bool
elem’ _ [] = False
elem’ e (x:xs) = e == x || elem’ e xs
```



```
ghci> sum' [1,2,3]
6
ghci> take' 2 [1,2,3]
[1,2]
ghci> last' [1,2,3]
3
ghci> and' [True,False,True]
False
ghci> concat' [[True],[False,True]]
[True,False,True]
ghci> replicate' 6 6
[6,6,6,6,6,6]
ghci> [1,2,3,4,5]Main.!!2
3
ghci> elem' 4  [1,2,3,4]
True
```

Question 6.7 : 7. Define a recursive function merge :: Ord a => [a] -> [a] -> [a] that merges two sorted lists to give a single sorted list.

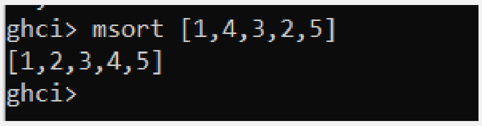
We can use recursion to do it, and base-case will be when either of any list is empty.
```
merge :: Ord a => [a] -> [a] -> [a]
merge a [] = a
merge [] b = b
merge (a:as) (b:bs)
    | a < b = a : merge as (b:bs)
    | otherwise = b : merge (a:as) bs
```



```
ghci> merge [1,2] [3,4]
[1,2,3,4]
```

Question 6.8 : Using merge, define a function msort :: Ord a => [a] -> [a] that implements merge sort, in which the empty list and singleton lists are already sorted, and any other list is sorted by merging together the two lists that result from sorting the two halves of the list separately.

```
merge :: Ord a => [a] -> [a] -> [a]
merge a [] = a
merge [] b = b
merge (a:as) (b:bs)
   | a < b = a : merge as (b : bs)
   | otherwise = b : merge (a : as) bs

halve :: [a] -> ([a], [a])
halve xs = splitAt ((length xs) ‘div‘ 2) xs

msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = merge (msort left) (msort right)
   where
       (left, right) = halve xs
```



```
ghci> msort [1,4,3,2,5]
[1,2,3,4,5]
ghci>
```

Question 6.9 : Using the five-step process, construct the library functions that:

a. calculate the sum of a list of number

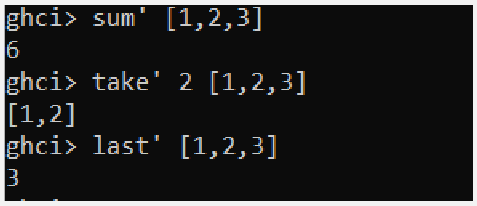
```
sum’ :: Num p => [p] -> p
sum’ [] = 0
sum’ (x:xs) = x + sum’ xs
```

b. take a given number of elements from the start of a list;

```
take’ :: Int -> [a] -> [a]
take’ 0 _ = []
take’ n (x:xs) = x : take’ (n - 1) xs
```

c. select the last element of a non-empty list.

```
last’ :: [a] -> a
last’ [x] = x
last’ (x:xs) = last’ xs
```



```
ghci> sum' [1,2,3]
6
ghci> take' 2 [1,2,3]
[1,2]
ghci> last' [1,2,3]
3
```

Question 7.1 : Show how the list comprehension [f x | x <- xs, p x] can be re-expressed using the higherorder functions map and filter.

map f (filter p xs)

Question 7.2 : Without looking at the definitions from the standard prelude, define the following higher-order library functions on lists.

a. Decide if all elements of a list satisfy a predicate: all :: (a -> Bool) -> [Bool] -> Bool

all p = and . map p

b. Decide if any element of a list satisfies a predicate: any :: (a -> Bool) -> [Bool] -> Bool

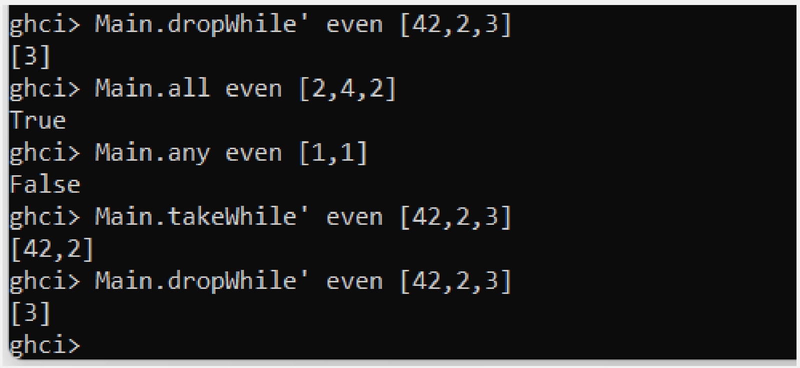
any p = or . map p

c. Select elements from a list while they satisfy a predicate: takeWhile :: (a -> Bool) -> [a] -> [a]

```
takeWhile p [] = []
takeWhile p (x:xs)
   | p x = x : takeWhile p xs
   | otherwise = []
```

d. Remove elements from a list while they satisfy a predicate: dropWhile :: (a -> Bool) -> [a] -> [a]

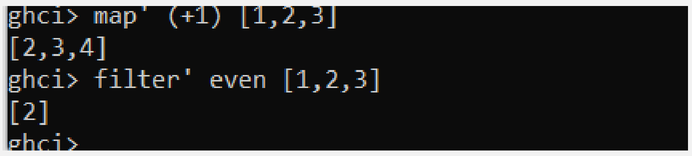
```
dropWhile p [] = []
dropWhile p (x:xs)
   | p x = dropWhile” p xs
   | otherwise = x : xs
```



```
ghci> Main.dropWhile' even [42,2,3]
[3]
ghci> Main.all even [2,4,2]
True
ghci> Main.any even [1,1]
False
ghci> Main.takeWhile' even [42,2,3]
[42,2]
ghci> Main.dropWhile' even [42,2,3]
[3]
ghci>
```

Question 7.3 : Redefine the functions map f and filter p using foldr.

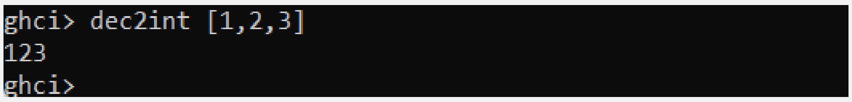
map f = foldr (xs -> f x : xs) []
filter p = foldr (xs -> if p x then x:xs else xs) []]]



```
ghci> map' (+1) [1,2,3]
[2,3,4]
ghci> filter' even [1,2,3]
[2]
ghci>
```

Question 7.4 : Using foldl, define a function dec2int :: [Int] -> Int that converts a decimal number into an integer.

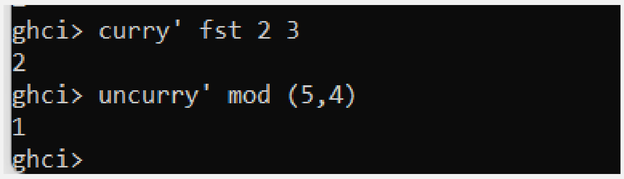
```
dec2int = foldl (y -> 10*x + y) 0
```



```
ghci> dec2int [1,2,3]
123
ghci>
```

Question 7.5 : Without looking at the definitions from the standard prelude, define the higher-order library function curry that converts a function on pairs into a curried function, and, conversely, the function uncurry that converts a curried function with two arguments into a function on pairs.

We will define two functions curry’ and uncurry’ like below, as functions curry and uncurry is already there, in prelude: curry’ :: ((a,b) -> c) -> (a -> b -> c)
curry’ f = y -> f (x,y)
uncurry’ :: (a -> b -> c) -> ((a,b) -> c)
uncurry’ f = (x,y) -> f x y



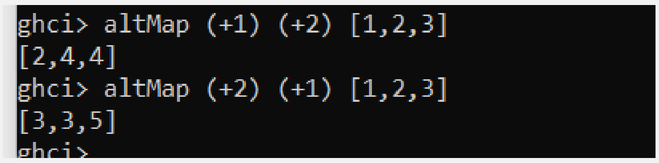
```
ghci> curry' fst 2 3
2
ghci> uncurry' mod (5,4)
1
ghci>
```

Question 7.6 : Redefine the functions chop8, map f and iterate f using unfold.

For this question code is given in separate file. There is no need to write it here too, as code is very lengthy.

Question 7.9 : Define a function altMap :: (a -> b) -> (a -> b) -> [a] -> [b] that alternately applies its two argument functions to successive elements in a list, in turn about order.

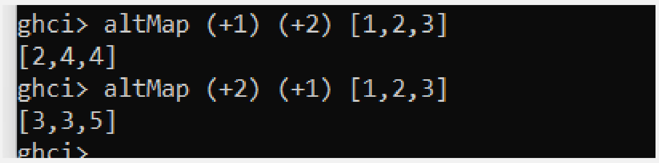
```
altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
altMap _ _ [] = []
altMap f0 f1 (x:xs) = f0 x : altMap f1 f0 xs
```



```
ghci> altMap (+1) (+2) [1,2,3]
[2,4,4]
ghci> altMap (+2) (+1) [1,2,3]
[3,3,5]
ghci>
```

Question 7.10 : Using altMap, define a function luhn :: [Int] -> Bool that implements the Luhn algorithm from the exercises in chapter 4 for bank card numbers of any length. Test your new function using your own bank card.

```
altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
altMap _ _ [] = []
altMap f0 f1 (x:xs) = f0 x : altMap f1 f0 xs

luhnDouble :: Int -> Int
luhnDouble x =
if x * 2 > 9
then x * 2 - 9
else x * 2

luhn :: [Int] -> Bool
luhn xs = sum (altMap luhnDouble id xs) ‘mod‘ 10 == 0
```



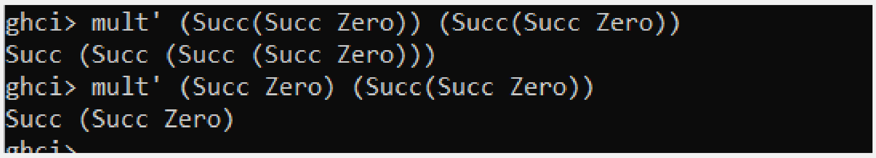
```
ghci> altMap (+1) (+2) [1,2,3]
[2,4,4]
ghci> altMap (+2) (+1) [1,2,3]
[3,3,5]
ghci>
```

Question 8.1 : In a similar manner to the function add, define a recursive multiplication function mult :: Nat -> Nat -> Nat for the recursive type of natural numbers:

Answer given in book is not very much runnable. We need to modify that. For, that we will define data type Nat recursively.

```
data Nat
   = Zero
   | Succ Nat
   deriving Show

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n
```

```
int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n - 1))

add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)

mult :: Nat -> Nat -> Nat
mult Zero _ = Zero
mult (Succ m) n = add n (mult m n)
```



```
ghci> mult' (Succ(Succ Zero)) (Succ(Succ Zero))
Succ (Succ (Succ (Succ Zero)))
ghci> mult' (Succ Zero) (Succ(Succ Zero))
Succ (Succ Zero)
```

Question 8.2 : Using this function, redefine the function occurs :: Ord a => a -> Tree a ->Bool for search trees. Why is this new definition more efficient than the original version?

This implementation is more efficient than guard because it uses less comparison for worse cases.

```
data Tree a
  = Leaf a
  | Node (Tree a) a (Tree a)

occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y) = x == y
occurs x (Node l y r) =
  case compare x y of
    LT -> occurs x l
    EQ -> True
    GT -> occurs x r
```

Question 8.3 : Consider the following type of binary trees: data Tree a = Leaf a | Node (Tree a) (Tree a) Let us say that such a tree is balanced if the number of leaves in the left and right subtree of every node differs by at most one, with leaves themselves being trivially balanced. Define a function balanced :: Tree a -> Bool that decides if a binary tree is balanced or not.

```
data Tree a
    = Leaf a
    | Node (Tree a) (Tree a)

balanced :: Tree a -> Bool
balanced (Leaf _) = True
balanced (Node l r) = abs (size l - size r) <= 1  balanced l  balanced r

size :: Tree a -> Int
size (Leaf _) = 1
size (Node l r) = size l + size r
```

Question 8.4 : Define a function balance :: [a] -> Tree a that converts a non-empty list into a balanced tree.

```
halve :: [a] -> ([a], [a])
halve xs = splitAt (length xs ‘div‘ 2) xs

data Tree a
    = Leaf a
    | Node (Tree a) (Tree a)
    deriving (Show)

balance :: [a] -> Tree a
balance [] = error "Empty list"
balance [x] = Leaf x
balance xs = Node (balance l) (balance r)
    where
        (l, r) = halve xs
```

Question 8.5 : Given the type declaration data Expr = Val Int | Add Expr Expr define a higher-order function folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a such that folde f g replaces each Val constructor in an expression by the function f, and each Add constructor by the function g.

```
data Expr
    = Val Int
    | Add Expr Expr

folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a
folde f _ (Val a) = f a
folde f g (Add a b) = g (folde f g a) (folde f g b)
```

Question 8.6 : Using folde, define a function eval :: Expr -> Int that evaluates an expression to an integer value, and a function size :: Expr -> Int that calculates the number of values in an expression

```
data Expr
   = Val Int
   | Add Expr Expr

folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a
folde f _ (Val a) = f a
folde f g (Add a b) = g (folde f g a) (folde f g b)

eval :: Expr -> Int
eval = folde id (+)

size :: Expr -> Int
size = folde (const 1) (+)
```

Question 8.7 : Complete the following instance declarations:
instance Eq a => Eq (Maybe a) where
...
instance Eq a => Eq [a] where
...

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x /= y = not (x == y)

instance Eq Bool where
   False == False = True
   True == True = True
   $_= =_= True$

instance Eq (Maybe a) where
   Nothing == Nothing = True
   Just x == Just y = x == y
   $_= =_= False$

instance Eq [a] where
   [] == [] = True
   (x:xs) == (y:ys) = x == y  xs == ys
   _ == _ = False
```

Question 8.8 : Extend the tautology checker to support the use of logical disjunction and equivalence ( ) in propositions.

For this question code is given in separate file. As, code is very large and there is no need to rewrite it here.