CS331: Assignment #3

Due on Wednesday, January 25, 2022

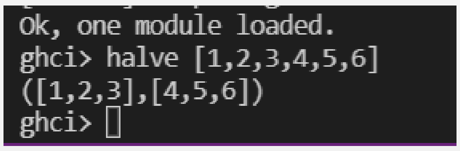
Siddharth Charan 190101085

Assignment: Introduction

In this assignment we were asked to solve questions given in exercise of chapter 4 and chapter 5 of book "Programming in Haskell". There are 8 questions in 4th chapter and 10 are in 5th chapter, which are discussed below.

Question 4.1: Using library functions, define a function halve :: [a] -> ([a],[a]) that splits an evenlengthed list into two halves. For example: > halve [1,2,3,4,5,6] = ([1,2,3],[4,5,6])

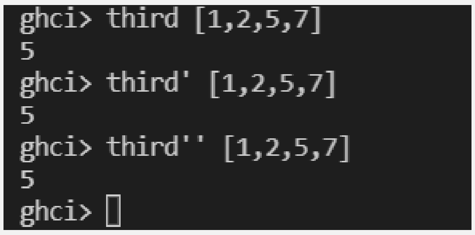
We can use library function "splitAt" to solve the purpose. It takes a location as input and splits the list at that particular location. We will give location as array/2 and we will have our desire splitted lists. Code is given in separate file. But, same can be found below:
halve :: [a] -> ([a], [a])
halve xs = splitAt ((length xs) ‘div‘ 2) xs



```
Ok, one module loaded.
ghci> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
ghci> 
```

Question 4.2: Define a function third :: [a] -> a that returns the third element in a list that contains at least this many elements using: a. head and tail; b. list indexing !!; c. pattern matching.

In this we will implement function third in 3 different ways, obeying instructions given in each part. Code is given in separate file. But, same can be found below:
(a) **third :: [a] -> a**
third xs = head (tail (tail (xs)))
(b) **third’ :: [a] -> a**
third’ xs = xs !! 2
(c) **third” :: [a] -> a**
third” (_:_:x:xs) = x



```
ghci> third [1,2,5,7]
5
ghci> third' [1,2,5,7]
5
ghci> third'' [1,2,5,7]
5
ghci> 
```

Question 4.3: Consider a function safetail :: [a] -> [a] that behaves in

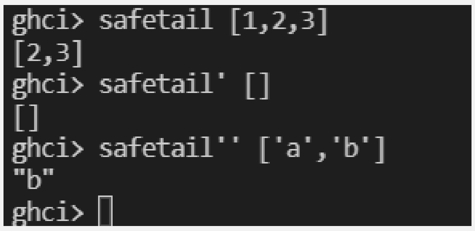
the same way as tail except that it maps the empty list to itself rather than producing an error. Using tail and the function null :: [a] -> Bool that decides if a list is empty or not, define safetail using:
a. a conditional expression;
b. guarded equations;
c. pattern matching.

We can modify tail to add the extra constrain. In this we will implement function third in 3 different ways, obeying instructions given in each part. Code is given in separate file. But, same can be found below:

(a)
```
safetail :: [a] -> [a]
  safetail xs =
    if null xs
      then []
      else tail xs
```

(b)
```
safetail’ :: [a] -> [a]
  safetail’ xs
    | null xs = []
    | otherwise = tail xs
```

(c)
```
safetail” :: [a] -> [a]
  safetail” [] = []
  safetail” (_:xs) = xs
```


```
ghci> safetail [1,2,3]
[2,3]
ghci> safetail' []
[]
ghci> safetail'' ['a','b']
"b"
ghci> 
```

Question 4.4: In a similar way to in section 4.4, show how the disjunction operator || can be defined in four different ways using pattern matching.

We can define as follow:

(a)
```
(||) :: Bool -> Bool -> Bool
  True || True = True
  True || False = True
  False || True = True
  False || False = False
```

(b)
```
(||) :: Bool -> Bool -> Bool
```

```
False || False = False
_|| _= True
```

(c) **(||) :: Bool -> Bool -> Bool**

```
 True || _= True
 False || a = a
```

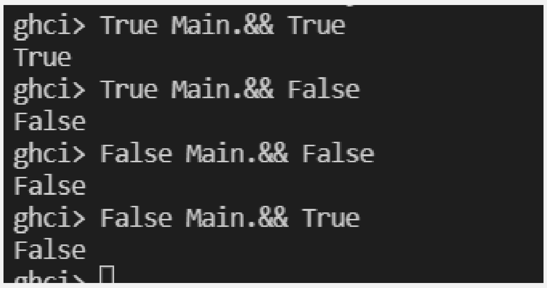
(d) **(||) :: Bool -> Bool -> Bool**

```
a || b
   | a == b = a
   | otherwise = True
```

Question 4.5: Without using any other library functions or operators, show how the meaning of the following pattern matching definition for logical conjunction can be formalised using conditional expressions: True True = True _ && _= False

Without using library function, we can simply implement above function using conditional logic. Code is given in separate file. But, same can be found below:

```
(&&) :: Bool -> Bool -> Bool
a && b =
     if a == True
       then if b == True
                 then True
                 else False
       else False
```

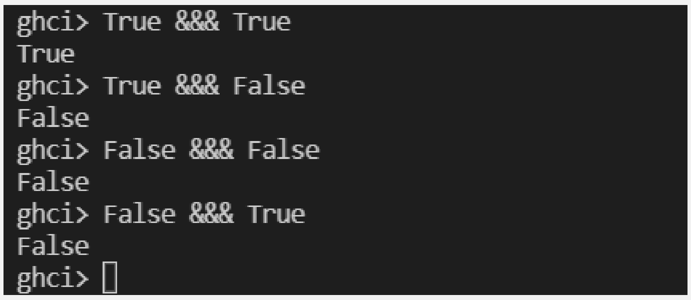


```
ghci> True Main.&& True
True
ghci> True Main.&& False
False
ghci> False Main.&& False
False
ghci> False Main.&& True
False
ghci> 
```

Question 4.6: Do the same for the following alternative definition, and note the difference in the number of conditional expressions that are required: True && b = True False && _= False

Here, we will again using conditional logic. Although for any input pair combination of true and false, operator'&&' gives same output as above, but due to change in definition, conditional implementation will also be changed. Code is given in separate file. But, same can be found below:

```
(&&) :: Bool -> Bool -> Bool
a && b =
  if a == True
    then b
    else False
```

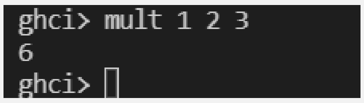


```
ghci> True &&& True
True
ghci> True &&& False
False
ghci> False &&& False
False
ghci> False &&& True
False
ghci> 
```

Question 4.7: Show how the meaning of the following curried function definition can be formalised in terms of lambda expressions:
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z

We can define above function as below:

```
mult :: Int -> Int -> Int -> Int
mult = \x -> (\y -> (\z -> x * y * z))
```



```
ghci> mult 1 2 3
6
ghci> 
```

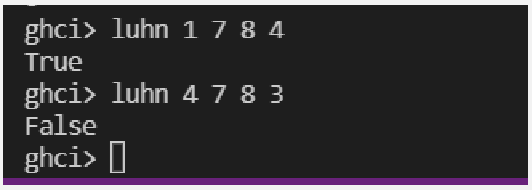
Question 4.8: Using luhnDouble and the integer remainder function mod, define a function luhn :: Int -> Int -> Int -> Int -> Bool that decides if a four-digit bank card number is valid.

We can define luhnDouble using conditional statement. As input number is 4 digit number and input is given as separate digits, we will take input in that particular order and using luhnDouble function, we can define desired luhn function as below, same code is given in separate file too.

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

luhn :: Int -> Int -> Int -> Int -> Bool
```

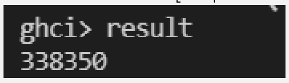
luhn a b c d = (luhnDouble a + b + luhnDouble c + d) ‘mod‘ 10 == 0



```
ghci> luhn 1 7 8 4
True
ghci> luhn 4 7 8 3
False
ghci> 
```

Question 5.1: Using a list comprehension, give an expression that calculates the sum of the first one hundred integer squares.

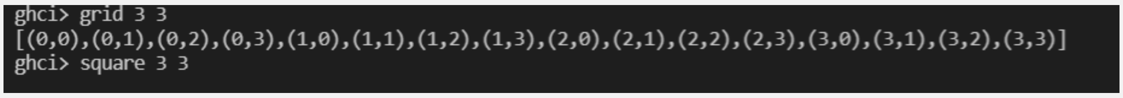
We can calculate sum of squares of first 100 numbers. It is 338350. We can write huskell program as below, to get same output:
result :: Integer
result = sum $[x^2|x<-[1..100]]$.



```
ghci> result
338350
```

Question 5.2: Suppose that a coordinate grid of size m $\times$ n is given by the list of all pairs (x, y) of integers such that Using a list comprehension, define a function grid :: Int -> Int-> [(Int,Int)] that returns a coordinate grid of a given size.

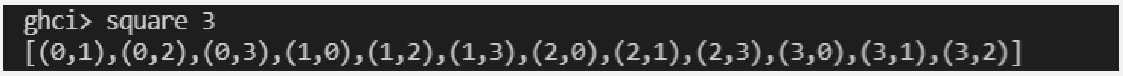
We can define grid as follows:
grid :: Int -> Int -> [(Int, Int)]
grid m n = [(x, y) | x <- [0 .. m], y <- [0 .. n]]



```
ghci> grid 3 3
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),(2,0),(2,1),(2,2),(2,3),(3,0),(3,1),(3,2),(3,3)]
ghci> square 3 3
```

Question 5.3: Using a list comprehension and the function grid above, define a function square :: Int -> [(Int,Int)] that returns a coordinate square of size n, excluding the diagonal from (0, 0) to (n,n).

Here, we just need to exclude the diagonal and hence, we can do it like below:
square :: Int -> [(Int, Int)]
square n = [(x, y) | x <- [0 .. n], y <- [0 .. n], x /= y]

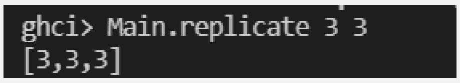


```
ghci> square 3
[(0,1),(0,2),(0,3),(1,0),(1,2),(1,3),(2,0),(2,1),(2,3),(3,0),(3,1),(3,2)]
```

Question 5.4: In a similar way to the function length, show how the library function replicate :: Int -> a-> [a] that produces a list of identical

elements can be defined using a list comprehension

We can define, replicate function as follows:
replicate :: (Num t, Enum t) => t -> a -> [a]
replicate n x = [x | _<- [1..n]]

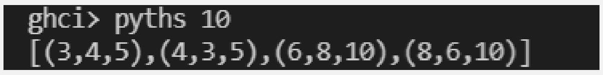


```
ghci> Main.replicate 3 3
[3,3,3]
```

Question 5.5: A triple (x, y, z) of positive integers is Pythagorean if it satisfies the equation x^2 + y^2 = z^2. Using a list comprehension with three generators, define a function pyths :: Int -> [(Int,Int,Int)] that returns the list of all such triples whose components are at most a given limit.

We can use brute-force technique to check all triplets of (x,y,z). If $x^2 + y^2 == z^2$, then only we will select those triplet. We can do it as follows:

```
pyths n =
   [ (x, y, z)
   | x <- [1 .. n]    , y <- [1 .. n]
   , z <- [1 .. n]
   , $x^2 + y^2 == z^2$
]
```



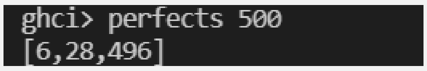
```
ghci> pyths 10
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

Question 5.6: A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension and the function factors, define a function perfects :: Int ->[Int] that returns the list of all perfect numbers up to a given limit.

We will calculate all factors of given number(except itsellf) and then we will calculate sum of all those factors and if sum is equal to given number than, we will include that number to output list. Implementation is given below as well as in separate file.

factors_except_itself :: Int -> [Int]
factors_except_itself n = [x | x <- [1 .. n-1], n ‘mod‘ x == 0]

perfects :: Int -> [Int]
perfects n = [x | x <- [1 .. n], x == sum (factors_except_itself x)]

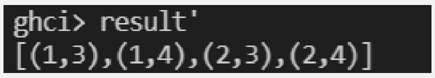


```
ghci> perfects 500
[6,28,496]
```

Question 5.7: Show how the list comprehension [(x,y) | x <- [1,2], y <-

[3,4]] with two generators can be re-expressed using two comprehensions with single generators.

We can use library function "concat :: [[a]] -> [a]" to do this. Implementation of this is given below:
result' = concat [[(x, y) | y <- [3, 4]] | x <- [1, 2]]

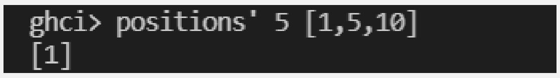


```
ghci> result'
[(1,3),(1,4),(2,3),(2,4)]
```

Question 5.8: Redefine the function positions using the function find.

Function position gives position of a particular element in list. We can define it as follows using function find:
find :: Eq a => a -> [(a, b)] -> [b]
find k t = [v | (k', v) <- t, k == k']

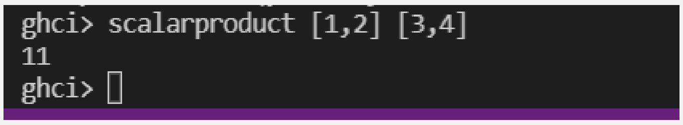
positions' :: Eq a => a -> [a] -> [Int]
positions' x xs = find x (zip xs [0 ..])



```
ghci> positions' 5 [1,5,10]
[1]
```

Question 5.9: show how a list comprehension can be used to define a function scalarproduct :: [Int] -> [Int] -> Int that returns the scalar product of two lists

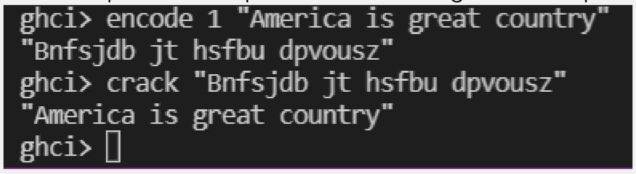
We can define scalarproduct function as follows:
scalarproduct :: Num a => [a] -> [a] -> a
scalarproduct xs ys = sum [x * y | (x, y) <- zip xs ys]



```
ghci> scalarproduct [1,2] [3,4]
11
ghci>
```

Question 5.10: Modify the Caesar cipher program to also handle uppercase letters.

For this particular question code is given in separate file.



```
ghci> encode 1 "America is great country"
"Bnfsjdb jt hsfbu dpvousz"
ghci> crack "Bnfsjdb jt hsfbu dpvousz"
"America is great country"
ghci>
```