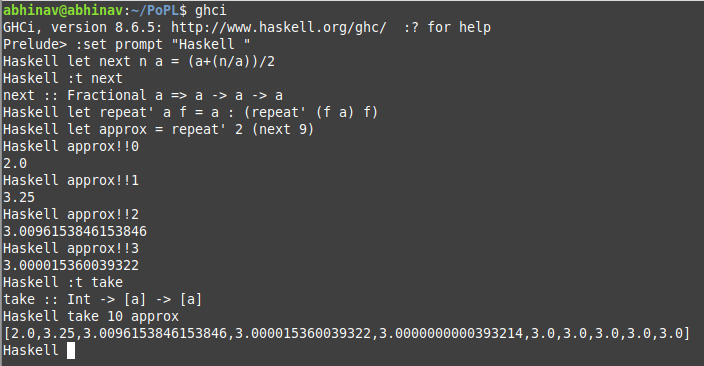
**19CSE313 Programming Paradigms**

**Practice Problems – Haskell**

1. Implement function(s) in Haskell to find the square root of a given number. The Newton-Raphson method to find the square root of a given number (*num*) is provided below for your reference. (Consider using recursion ;-)
   1. Take a reasonable *guess* (which in fact is an approximate square root - 1.0 is a good place to start) for the square root.
   2. Find the average of *guess* and *num/guess* (Note that this step keeps improving the guess such that the guess which is the approximate square root moves towards the precise square root)
   3. Repeat step b. if the approximate square root (I.e. *guess*) is not within the tolerance limit.

**Code:**



1. In your school days you must have studied about *singular* and *plural* nouns. You must also have studied various rules for making *plural* nouns given a *singular* noun. You, by now, must be aware that even those rules have exceptions. Please find a file, in this folder, named *plurals\_rule\_chart.pdf*. The rule chart provides some well-known rules to make plurals. (Of course, there are exceptions to these rules which you can ignore for now). Write a function(s) in Haskell, that accepts a noun and makes an appropriate plural as dictated by the rules provided in the file.
2. Write functions in Haskell to compute complex number (arithmetic) operations. Please find a file, in this folder, named *Complex Math Formulas.pdf*. The file lists various operations that you can perform on complex number(s).
3. In the following problems you have been asked to define/implement several functions. You must use **list comprehensions**.
   1. Write a function halfEvens :: [Int] -> [Int] that returns half of even even number in the given list. For example,

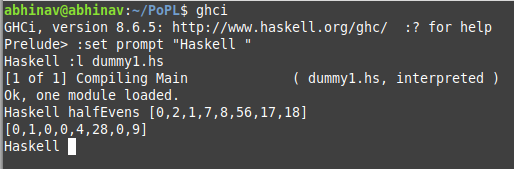
* halfEvens [0,2,1,7,8,56,17,18] should give [0,1,4,28,9]

**Code:**

halfEvens :: [Int] -> [Int]

halfEvens = map (\x -> if (x `mod` 2) == 0 then x `div` 2 else x-x)

**Output:**



* 1. Write a function inRange :: Int -> Int -> [Int] -> [Int] to return all numbers in the input list within the range given by the first two arguments (inclusive). For example,

inRange 5 10 [1..15] should give [5,6,7,8,9,10]

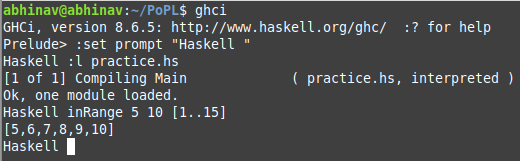
InRange 5 10 [7..15] should give [7,8,9,10]

**Code:**

inRange :: Int -> Int -> [Int] -> [Int]

inRange lo hi xs = [ x | x <- xs, x >= lo && x <= hi ]

**Output:**

****

* 1. Write a function multiDigits :: String -> Int that returns the product of all the digits in the input string. If there are no digits in the input string, your function should return 1. For example,

multiDigits “The time is 4:25” should give 40

multiDigits “No digits here” should give 1

You’ll need a library function to determine if a character is a digit and a library function to convert a digit to an integer.

* 1. Write a function countDigits :: String -> Int that returns the number of digits in the input string. For example,

countDigits “The time is 4:25” should give 3

countDigits “No digits here” should give 0

* 1. Write a function captialize :: String -> String which, given a word, capitalizes the first character. For example,

capitalize “coIMBAtore” should give “Coimbatore”

* 1. Write a function search :: String -> Char -> [Int] that returns the positions of all occurrences of the second argument in the first. For example,

search “Bookshop” ‘o’ should give [1,2,6]

search “senselessness” ‘s’ should give [0,3,7,8,11,12]

1. The foldr function is defined as follows.

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f b [] = b

foldr f b (x:xs) = f x (foldr f b xs)

foldr gives the abstract pattern of recursion over lists, and it can be used to give abstract definitions of many functions that operate on lists.

Define the following functions in terms of foldr - and, or, length, map, reverse and ++.

* foldr appendright [] = reverse
* foldr (++) [] : Dissolves one level of brackets