```haskell
--                                       Home Work 1

-- Name : Anupkumar Nagaraj Joshi
-- UNM ID : 101880602
-- email : anupknjoshi@unm.edu
-- Date : 5/Sept/2019


import Data.List
import System.IO

--[1]
--a)
interleave :: [a] -> [a] -> [a]
interleave xs [] = xs --checks for empty list condition
interleave  []  ys  = ys --checks for empty list condition
interleave (x:xs) (y:ys) = x : y : interleave xs ys

--input : interleave [1,2,3] [4,5,6,7,8]
-- output : [1,4,2,5,3,6,7,8]

--b)
heads :: [[a]] -> [a]
--First pattern. If the list is empty return a empty list.
heads [] = []  --checks for empty list condition
--Catch all pattern. Receive a list of lists (xxs). For every inner list (xs <-
xxs) call the head method (head xs), but only when the inner list is not null or
empty (not(null xs).
heads xxs = [head xs | xs <- xxs, not(null xs)]

--input : heads [[1,2],[],[3,4,5]]
--output : [1,3]

--c)
tails1 :: [[a]] -> [[a]]
tails1 [] = [] --checks for empty list condition
tails1 xxs = [  tail xs | xs <- xxs, not(null xs)]

--input : tails1 [[1,2],[],[3,4,5]]
-- output :  [[2],[4,5]]

--d)
interleaveN :: [[a]] -> [a]
interleaveN x = concat(transpose x) -- transposes before concatination

--input : interleaveN [[1,2],[],[3,4,5]]
--output : [1,3,2,4,5]

--[2]

--a)
doubleEach :: [Int] -> [Int]
doubleEach = map (\x -> x+x) -- double formula

--input : doubleEach [1,2,3]
--output :  [2,4,6]

--b)
doubleEach' :: [Int] -> Int
doubleEach' x =  foldr (*) 2 x -- returns sum of dooubles

--input : doubleEach [1,2,3]
--output : 12
```

```
filterEven ::  [Int] -> [Int]
filterEven  = filter even

--c)
doubleEven :: [Int] -> [Int]
doubleEven x =  doubleEach(filterEven x)  -- doubles and then filters

--input : doubleEven [1,2,3,4]
--output :  [4,8]


--[3]
--a)
takeUntil :: (a -> Bool) -> [a] -> [a]
takeUntil _ [] = []
takeUntil p (x:xs) =  if p x then [] else x : takeUntil p xs  -- takes until
condition is satisfied

--input : takeUntil (\x -> x `mod` 2 == 0) [1,3,4]
--output :  [1,3].

--b)
allSquares :: [Int]
allSquares  = [  x^2 | x <- [1..]]  -- gives square of the numbers till infinity

--input : nothing
--output : [1,4,9..........]

--c)

finalFunction :: Int
finalFunction = sum(takeUntil (\x -> x == 100) allSquares) --takes sum of
squares until predicate is true

--input : nothing
--output : 285
```