Lesson 4

List comprehension and recursive functions examples

- [x^2 | x <- [1..5]]

[1,4,9,16,25]

x <-[1..5] is a generator

- [(x,y) | x<-[1,2,3], y <-[4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

> [(x,y) | y <-[4,5], x<-[1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
like two nested loops

- [(x,y) | x <- [1..3], y <- [x..3]]

[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

list comprehension can also use gards

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

prime :: Int -> Bool
prime n = factors n == [1,n]

```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]

zip :: [a] -> [b] -> [(a,b)]
zip [] y = []
zip x [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

- ➢ pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]

```
sorted : [a] -> Bool
sorted xs = and [x<=y | (x,y) <- pairs xs]

positions : Eq a => a  -> [a] -> [Int]
positions x xs = [i | (x’,i’) <- zip xs [0..], x == x’]

>positions False [True, False, False]
[1,2]
```

String comprehension

Strins are lists of Char, hence list operators and list comprehension applies to them

’’abcde’’ !! 3
‘c’

length ’’abcde’’
5

take 3 ’’abcde’’
’’abc’’

zip ’’abc’’ [1,2,3,4]
[(‘a’,1),(‘b’,2),(‘c’,3)]

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, x >=‘a’ && x <= ‘z’]

count :: Char -> String ->  Int
count  x xs = length [x’ | x’ <-xs, x == x’]


cout ‘s’ ‘’sassuolo’’
3
```

Exercises (chapter 5)

4) define function replicate :: Int -> a -> [a]
replicate 3 True
[True,True,True]

5)A triple (x,y,z) is Pythagorean when x^2 + y^2 ==z^2, write a function that computes a list of all Pythagorean triples whole elements are at most a given n

6) A positive integer is perfect if it equals the sum of its factors (excluding the number self), define perfects :: Int ->[Int] that, given n, computes the list of all perfect numbers in [1..n]

7) Show that [(x,y) | x <- [1,2], y <-[3,4]] can be re-expressed with 2 comprehensions with single generators

recursive functions

advice on recursion

1. define the type as a postcondition: before you write the function
2. enumerate the cases unserstand how many are needed
3. define the simpler cases
4. define the other cases
5. generalize and simplify

drop

1. drop :: Int -> [a] -> [a]
2. enumerate cases

```
drop 0 []        =
drop 0 (x:xs)    =
drop n []        =
drop n (x:xs)    =
```

3. define simple cases

```
drop 0 []        = []
drop 0 (x:xs)    = (x:xs)
drop n []        = []
```

4. define other cases

```
drop n (x:xs) =  drop (n-1) xs
```

5. simplify and generalize

```
drop :: Integral b => b -> [a] -> [a]

drop 0 []          = []              ➔ drop 0 xs = xs
drop 0 (x:xs)      = (x:xs)
drop n []          = []              ➔ drop _ [] = []
drop n (x:xs)      = drop (n-1) xs   ➔ drop n (_:xs) = drop (n-1)  xs
```

Exercise 9

using the 5 steps process construct the library functions

.sum

.take

.last

main program of Homework 1 with IO

```
module Homework1 where

import Validate
import System.IO

main :: IO()
main = do
      logLines<- mainloop
      print (map (validate.read)  logLines)

mainloop :: (IO [String])
mainloop  =  do inpStr <- getLine
              if inpStr == "end"
                        then return []
                else do listLines <- mainloop
                      return (inpStr : listLines)
```

DO used where is IO

main use inpure I/O

an call ather functions thar are pure

IO cannot be done outside main

validate composed with read

return is the opposite of <-

fromally return takes a value and puts it into a IO box
Arroaw takes out of a IO box