Lesson 4

List comprehension

recursive functions

higher order 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], primes x]

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

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

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

Exercises (chapter 5)

3) 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

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
2. enumerate the cases
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

Higher order functions

functions that return functions as result

obvious with currying

functions that take functions as parameters

```
twice :: (a -> a) -> a -> a
twice f = f . f
```

```
twice (*2) 3
12
```

```
twice reverse [1,2,3]
[1,2,3]

map :: (a -> b) -> [a] -> [b]
map f xs = [ fx | x <- xs]

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

map reverse [″abc ″, ″ def ″]
[″ cba ″, ″fed ″]
```

two maps to work on nested list

```
map (map (+1)) [[1,2,3],[4,5]]
={applying outer map}
[map (+1) [1,2,3], map (+1) [4,5]]
={applying inner maps}
[[2,3,4],[5,6]]
```

```
filter :: (a -> Bool) -> [a] ->  [a]
filter p xs = [x | x <- xs, p x]
filter even [1..10]
[2,4,6,8,10]
```

```
filter (\= ` `) ″abc   def  ghi″
″abcdefghi″

filter p []                              = []
filter p (x:xs) | px                     = x : filter p xs
                | otherwise = filter p xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []      = v
foldr f v (x:xs) = f x (foldr f v xs)
```

```
map f = foldr ((:) . f) []
```

surprising: reverse a list with foldr

```
snoc x xs = xs ++ [x]
reverse = foldr snoc []
```

```
foldr (#) v [x0,x1,...xn] = x0 # (x1 # (...(xn # v)...))
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

foldl (#) v [x0, x1,...,xn] = (...((v # x0) # x1)...) # xn

associates to the left

we can do reverse also with foldl:

```
reverse = foldl (\xs -> \ x -> x:xs) []
```

```
map f = foldl (\v -> (\x -> v ++ [f x])) []
```

the function composition operator

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

identity for composition:

```
id : a -> a
id = \x -> x
```

```
compose :: [a->a] -> (a->a)
compose = foldr (.) id
```

Exercise 2 c

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p  [] = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs
                                        else []
```