Lesson 5

Higher-order functions

Higher order functions

- functions that return functions as result

obvious with curryfication

- 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 lists of lists :: [[a]]

```
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
```

From the article «Why functional programming matters» by John Hughes

The importance of modularity and «gluing» functions together

sum [] = $\boxed{0}$
sum (x:xs)= x $\boxed{+}$ sum xs

sum can be modularized by gluing together a general recursive pattern and the boxed parts the recursive pattern is called foldr

sum = foldr (+) 0 generalized to   foldr f v

parameterizing sum:

```
sum [] = 0
sum (x:xs)= x + sum xs
```

```
(foldr f v) [] = v
foldr f v (x:xs)= f x ((foldr f v) xs)
```

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

product = foldr (*) 1

anyTrue = foldr (||) False

allTrue = foldr (&&) True

it replaces all : in a list by f and [] with 0/1/False/True

foldr (+) 0 (1:2:3:[])

1+(2+(3+0)) e in generale:

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

```
foldr (:) []  simply copies a list

append a b = foldr (:) b a

(:) replaced by (:) and [] by b


length = foldr count 0
count _ n = n + 1
```

surprising: reverse a list with foldr

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

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

if we want to double all elements of a list:

```
doubleAll:: Num a  => [a] -> [a]
doubleAll = foldr doubleAndCons []
```

where

```
doubleAndCons n xs = (n*2) : xs
```

doubleAndCons can be modularized further:

```
doubleAndCons = fAndCons double
where double n = 2 * n     and
fAndCons f x xs = (f x) : xs  = (:) (fx) xs
fAndCons  f  = (:) . f
```

e alla fine:

```
doubleAll= foldr (:).double []
```

but we know that doubleAll can be defined also with map:

doubleAll = map double

this is not by chance

map f = foldr (:).f Nil

map always has (:), foldr does not, is more general the same idea applies to other data structures (not just to lists)

data Tree a = Node a [Tree a] -- Node is a constructor, i.e. a function that builds the values of type Tree a:

Node :: a -> [Tree a] -> Tree a

leafs have [], i.e. empty list of subtrees

trees are built with Node, (:) and [], 3 things, hence:

foldtree f g a   Node / [Trees] / []

```
foldtree f g a (Node x tx) = f x (foldtree f g a tx)

foldtree f g a (t:tx)          = g (foldtree f g a t) (foldtree f g a tx)

foldtree f g a []              = a

sumtree = foldtree (+) (+) 0

labels = foldtree (:) append Nil

maptree = foldr (Node . f) (:) Nil
```

fold from left to right

```
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

```
sum = foldl (+) 0
```

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 []
```

Exercise 2 d: dropWhile

Exercise 3 redefine filter p using foldr