

# Lesson 4

List comprehension

recursive functions

higher order functions

# examples

➤  $[x^2 \mid x \leftarrow [1..5]]$   
[1,4,9,16,25]

$x \leftarrow [1..5]$  is a generator

this is the Outer one

➤  $[(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]]$   
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

>  $[(x,y) \mid y \leftarrow [4,5], x \leftarrow [1,2,3]]$   
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]  
like two nested loops

➤ `[(x,y) | x <- [1..3], y <- [x..3]]` ← like nested loops

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

1 - extract a  
the lists from the list of list  
2 - extract the element from list

`concat :: [[a]] -> [a]`

`concat xss = [x | xs <- xss, x <- xs]`

If the argument has a structure you can use pattern in generators

`length xs = sum [1 | <- xs]`

list comprehension can also use **guards**

`factors :: Int -> [Int]`

`factors n = [x | x <- [1..n], n 'mod' x == 0]`

`prime :: Int -> Bool`


`prime n = factors n == [1,n]`

very inefficient

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

the Type of a function tell us a lot of things (USEFUL)

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

 if one of the list is empty the result will be empty

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

tail eliminates the first element

```
➤ pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

the type must be Ord because we are using <=

and accept as parameter a list of Bool

`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  $[(x,y) \mid x \leftarrow [1,2], y \leftarrow [3,4]]$  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.  $\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]$

2. enumerate cases

$\text{drop } 0 [] =$

$\text{drop } 0 (x:xs) =$

$\text{drop } n [] =$

$\text{drop } n (x:xs) =$

3. define simple cases

$\text{drop } 0 [] = []$

$\text{drop } 0 (x:xs) = (x:xs)$

$\text{drop } n [] = []$

4. define other cases

$\text{drop } n \ (x:xs) = \text{drop } (n-1) \ xs$

5. simplify and generalize

$\text{drop} :: \text{Integral } b \Rightarrow b \rightarrow [a] \rightarrow [a]$

$\text{drop } 0 \ [] = [] \quad \rightarrow \text{drop } 0 \ xs = xs$

$\text{drop } 0 \ (x:xs) = (x:xs)$

$\text{drop } n \ [] = [] \quad \rightarrow \text{drop } \_ \ [] = []$

$\text{drop } n \ (x:xs) = \text{drop } (n-1) \ xs \quad \rightarrow \text{drop } n \ (\_:xs) = \text{drop } (n-1) \ xs$

## 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

$\text{twice} :: (a \rightarrow a) \rightarrow a \rightarrow a$

$\text{twice } f = f . f$

$\text{twice } (*2) 3$

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)...))`



$\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

$\text{foldl } f \ v \ [] = v$

$\text{foldl } f \ v \ (x:xs) = \text{foldl } f \ (f \ v \ x) \ xs$

$\text{foldl } (\#) \ v \ [x_0, x_1, \dots, x_n] = (\dots((v \# x_0) \# x_1) \dots) \# x_n$

# associates to the left

we can do reverse also with foldl:

$\text{reverse} = \text{foldl } (\backslash xs \rightarrow \backslash x \rightarrow x:xs) \ []$

$\text{map } f = \text{foldl } (\backslash v \rightarrow (\backslash x \rightarrow v ++ [f \ x])) \ []$

the function composition operator

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$f \cdot g = \lambda x \rightarrow f (g x)$$

identity for composition:

$$\text{id} : a \rightarrow a$$
$$\text{id} = \lambda x \rightarrow x$$
$$\text{compose} :: [a \rightarrow a] \rightarrow (a \rightarrow a)$$
$$\text{compose} = \text{foldr } (\cdot) \text{id}$$

## Exercise 2 c

`takeWhile :: (a -> Bool) -> [a] -> [a]`

`takeWhile p [] = []`

`takeWhile p (x:xs) = if p x then x : takeWhile p xs  
else []`