Lesson 3

defining functions

In this lesson we see many ways to define functions

1. composition of old functions to define new ones
even :: Integral a => a -> Bool
even n = n 'mod' 2 ==0

splitAt :: Int ->[a] ->([a],[a])
splitAt n xs =(take n xs, drop n xs)

2. Conditional

abs :: Int -> Int

abs n = in n >=0 then n else –n

signum :: Int -> Int

signum n = if n<0 then -1 else

              if n==0 then 0 else 1

always   if then exp1 else exp2

exp1:: T   exp2 :: T

no dangling else

3. Guarded equations

```
abs n | n >= 0 = n
      | otherwise =-n
```

easer to read when there are more then 2 choices

```
signum n | n<0 = -1
         | n==0 = 0
         | otherwise = 1
```

4. Pattern matching

not :: Bool -> Bool

not False = True

not True = False

(&&) :: Bool ->Bool ->Bool

True && True = True

True && False = False

False && True = False

False && False = False

True && True = True

_ && _ = False

True && b = b

False && _ = False

wrong:

b && b = b

_ && _= False

but one can write :

b && c | b == c   = b
          | otherwise =False

Tuple patterns
fst :: (a,b) -> a
fst (x,_)=x

snd : (a,b) -> b
snd (_,y) = y

List patterns

```
test :: [Char]-> Bool
test ['a',_,_] = True
test _           = False
```

[1,2,3] = Cons(1, (Cons 2, (Cons 3,[]))) =
1 : (2 : (3 : []))

```
test :: [Char] -> Bool
test ('a' : _) = True
test _           = False
```

Lambda expressions

\x -> x+x

they express naturally curried functions:

add:: Int -> Int -> Int
add = \x -> (\y -> x+y)

const function
const :: a -> b-> a
const x _ = x
const x = _ -> x

```
odds :: Int -> [Int]
odds n = map f [0..n-1]
           where f x = x*2+1


map:: (a->b) -> [a] -> [b]
map _  [] = []
map f (x: xs) = f x : map f xs


odds n = map (\x -> x*2+1) [0..n-1]

function with no name used only locally
```

operator sections
 infix functions, like +, are called operators

any binary function ca be used infix with 'op', like 7 'div' 2

but we can do also the opposite (+) is a binary function
(+) 2 3
5
and also
(2+) 3 = 5
(+3) 2 =5
(+) :: Int -> Int -> Int

if # is an operator, then (#), (x #) and (# y) are called **sections**

(1 +) = \x -> 1+x
(1/) = \x -> 1/x
(/2) = \x -> x/2

sections are necessary when asking the type

useful in espressions such as;
dl :: [Num]->[Num]
dl xs = map (2*) xs

Exercises

2) define the function third in 3 ways:

a) with head and tail

b) with !!

c) pattern matching

3) define safetail :: [a]->[a] that behaves like tail exceprt that with [], insted of failing, answers []. We can use tail and null :: [a] -> Bool that answers True iff the input is an empty list, define safetail using:

a) a conditional expression

b) guarded equations

c) pattern matching