Lesson 5

Declaring types and classes

3 type definitions:

1.**type** ➔ Transparent types: a new name for already existing types

2. **data** ➔ Opaque types, user defined types

3. **newtype** ➔ Especially simple opaque types: more efficiently implemented

1. Type

type String = [Char]

type Pos = (Int , Int)

type Trans = Pos -> Pos

type Tree = (Int, [Tree]) --not allowed, recursive!!

polymorphic : type Pair a = (a,a)

```
type Assoc k v = [(k,v)]

find :: Eq k => k -> Assoc k v -> v
find k t = head [v  | (k',v) <- t, k==k']
```

2. Data

from Prelude,

```
data Bool = False | True
```

False and True are constructors

`data A = B | C`   would also work

```
data Move = North | South | East | West
                  deriving Show
```

```
move :: Move -> Pos -> Pos

move North (x,y) = (x,y+1)
move South (x,y) = (x, y-1)
move East  (x,y)   = (x+1,y)
move West (x,y)   = (x-1,y)

moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)
```

esercizio invmoves ??

a more advanced example

data Shape = **Circle** Float | **Rect** Float Float

Circle and Rect are costructor functions:

- :t Rect

Rect :: Float -> Float -> Shape

>Rect 2.3 1.4

Rect 2.3 1.4

insomma il risultato di applicare Rect 2.3 1.4 è l’espressione stessa. The values of Shape are Rect 2.3 1.4 and Circle 3.3

```
square :: Float -> Shape
square x = Rect x x

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x*y

data Maybe  a = Nothing | Just a
```

is used to represent a success/failure behaviour of functions

Nothing stands for failure
Just x stands for success with result x

divisione safe

```
safediv : Int -> Int -> Maybe Int
safediv _ 0  = Nothing
safediv m n = Just (m ‘d’ n)

safe head :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

3. newtype

when a data definition has one costructor only with one parameter only then it is possible to define it like this,

```
newtype Nat = N Int
```

how does it compare with

```
type Nat = Int
data Nat = N Int  ??
```

Recursive types

types defined using data and newtype can be rcursive

```
Nat = Zero | Succ Nat

Zero, Succ Zero, Succ (Succ Zero), ....

nat2int :: Nat -> Int
nat2int Zero =0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

```
add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int n + nat2int m)
```

ma anche

```
add Zero n = n
add (Succ m)  n = Succ (add m n)
```

binary trees

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)

flatten :: Tree a -> [a]
flatten  (Leaf x) = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r

data Tree a = Node a [Tree a]
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
```

Class and instance declarations

a new class can be defined with the class mechanism

```
class Eq a where
(==), (/=) :: a -> a -> Bool
x /= y = not (x == y)
```

for a type a to be in Eq it must support == and /=, and is enough to define ==

```
instance Eq Bool where
    False == False = True
    True == True   = True
     _    == _     = False
```

only types defined by data and new types can me made into instances of classes

default definitions of methods (such as /=) can be overridden.

Classes can be extended to form new classes

```
class Eq a =>  Ord a where
(<), (<=), (>), (>=) :: a -> a -> Bool
min, max             :: a -> a -> a

min x y | x <= y        = x
        | otherwise  = y

max x y |  x <= y        = y
        | otherwise  = x
```

if we have a type in Eq, to make it also in Ord one needs to define 4 methods

```
instance Ord Bool where
    False < True = True
    _      <  _      = False
    b <= c = (b <c) || (b == c)
    b > c    = c < b
    b >= c  = c <= b
```

Derived instances

when new types are defined it is often convenient to make them into instances of built-in classes

data Bool = False | True
                    deriving (Eq, Ord, Show, Read)

in this way all metods that belong to the classes can be used for Bool:

- ➢ False == False

True

- ➢ False < True

True

the order False < True is determined by the declaration
data Bool = False | True

in case of costructors with arguments, the types of those arguments must be in the classes

data Shape = Circle Float | Rect Float Float

data Maybe a = Nothing | Just a

if me make Shape a deriving Class then Float in Class lexigographical order

- Rect 1.0 4.0 < Rect 2.0 3.0

True

- Rect 1.0 4.0 < Rect 1.0 3.0

False

> Circle 1.0 < Circle 1.1

True