

Lesson 6

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]) --recursive! Not allowed!!

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
```

$\text{move} :: \text{Move} \rightarrow \text{Pos} \rightarrow \text{Pos}$

$\text{move North } (x,y) = (x,y+1)$

$\text{move South } (x,y) = (x, y-1)$

$\text{move East } (x,y) = (x+1,y)$

$\text{move West } (x,y) = (x-1,y)$

$\text{moves} :: [\text{Move}] \rightarrow \text{Pos} \rightarrow \text{Pos}$

$\text{moves [] } p = p$

$\text{moves } (m:ms) p = \text{moves } ms (\text{move } m p)$

esercizio invmoves ??

a more advanced example

```
data Shape = Circle Float | Rect Float Float
```

Circle and Rect are constructor 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.

for example Rect 2.3 1.4 and Circle 3.3 are possible values of
type Shape


```
square :: Float -> Shape
```

```
square x = Rect x x
```

```
area :: Shape -> Float
```

```
area (Circle r) = pi * r^2
```

```
area (Rect x y) = x*y
```

partial operators and functions

`div : Int -> Int -> Int`

`10 `div` 0`

`* error «division by 0»`

`head []`

`* error «head on []»`

solution

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 constructor 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 recursive

$$\text{Nat} = \text{Zero} \mid \text{Succ Nat}$$

Zero, Succ Zero, Succ (Succ Zero),

$$\text{nat2int} :: \text{Nat} \rightarrow \text{Int}$$
$$\text{nat2int Zero} = 0$$
$$\text{nat2int (Succ } n) = 1 + \text{nat2int } n$$
$$\text{int2nat} :: \text{Int} \rightarrow \text{Nat}$$
$$\text{int2nat } 0 = \text{Zero}$$
$$\text{int2nat } n = \text{Succ (int2nat (n-1))}$$

$\text{add} :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$

$\text{add } m \ n = \text{int2nat } (\text{nat2int } n + \text{nat2int } m)$

ma anche

$\text{add } \text{Zero } n = n$

$\text{add } (\text{Succ } m) \ n = \text{Succ } (\text{add } m \ n)$

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

$(==), (/=) :: a \rightarrow a \rightarrow \text{Bool}$

$x /= y = \text{not } (x == y)$

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

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

only types defined by data and new types can be 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 let it belong also to 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 type classes

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

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

➤ False == False

True

➤ False < True

True

➤ show True

«True»

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

in case of constructors 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 we define `Shape` «deriving from Class X» then `Float` must be in Class X

lexicographical 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

➤ Circle 1.0 < Rect 2.0 3.0

True