# Haskell in razredi tipov

prejšnjič...

# **Fleksibilnost tipov** lahko zagotovimo na več načinov

## parametrični polimorfizem

**generiki** (Java, Rust), **polimorfizem** (OCaml, Haskell), ...

## ad-hoc polimorfizem

**značilnosti**/*traits* (Rust), **razredi tipov** (Haskell), ...

## podtipi

implicitne **pretvorbe**, **moduli** (OCaml), **podrazredi**, ...

# Vrednosti **manjšega** tipa so kompatibilne z **večjim**

$$\frac{\Gamma \vdash e : A \qquad A \leq B}{\Gamma \vdash e : B}$$

# Spoznali smo **pravila za določanje** podtipov

$$\frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{A_1 \times A_2 \leq B_1 \times B_2}$$

$$\frac{A_1 \geq B_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

# **Zapisni tipi** imajo bogato strukturo podtipov

$$\frac{\forall i \leq m \,.\, \exists j \leq n \,.\, \ell'_i = \ell_j \wedge A_j \leq B_i}{\{\ell_1 : A_1; \ldots; \ell_n : A_n\} \leq \{\ell'_1 : B_1; \ldots; \ell'_m : B'_m\}}$$

# **Spremenljivi zapisi** podtipov ne podpirajo

$$A = \{\texttt{mutable}\ x : \texttt{int}\}$$

$$B = \{\texttt{mutable}\ x : \texttt{float}\}$$

## branje polja

$\texttt{let}\ r_A(p : A) = p\,.x + 10$

$r_A(\{x : 3.14\})$ ✗

$\texttt{let}\ r_B(p : B) = p\,.x + 3.14$

$r_B(\{x : 10\})$ ✓

## pisanje v polje

$\texttt{let}\ w_A(p : A) = p\,.x \leftarrow 10$

$w_A(\{x : 3.14\})$ ✓

$\texttt{let}\ w_B(p : B) = p\,.x \leftarrow 3.14$

$w_B(\{x : 10\})$ ✗

# **Objekti** so bolj ali manj rekurzivni zapisi

| | |
|---|---|
| **konstruktorji & metode** | funkcije |
| **dedovanje** | nominalni podtipi |
| **prekrivanje** (*overriding*) | senčenje |
| **preobteževanje** (*overloading*) | ad-hoc polimorfizem |
| **generične metode & razredi** | parametrični polimorfizem |
| **abstraktne/virtualne metode & razredi** | specifikacije |
| **vmesniki** | specifikacije |
| **enkapsulacija** | modularnost & abstrakcija |

# Pri razredih imamo **dve pristopa** k podtipom

```
class C { int x; }
class D { int x; float y; }
class E extends C { float y; }
```

## nominalni pristop

Podrazredi so tisti, ki jih programer **navede**.

$E \leq C \quad D \not\leq E$

## strukturni pristop

Podrazredi so vsi s **kompatibilno strukturo**.

$E \leq C \quad D \leq E \quad E \leq D$

# nominalni in strukturni

# podtipi

## v OCamlu

# Uvod v Haskell

## **Aritmetične** operacije so večinoma standardne

In [1]:
```
12 * (34 + 67) - 89
```

In [2]:
```
22 / 7
```

```
3.142857142857143
```

In [3]:
```
12 ** 34
```

```
4.9222352429520267040371132431220080064
```

In [4]:
```
12 ^ 34
```

```
49222352429520267040371132431220080064
```

## **Logične operacije** so standardne

In [5]:
```
False && not (False || True)
```

**Evaluate**

**Found:** **Why Not:**

`False && not (False || True)`False

**Evaluate**

**Found:** **Why Not:**

`False || True`True

```
False
```

In [6]:
```
if True then 10 else 20
```

```
10
```

## **Primerjave** so standardne

In [7]:
```
1 == 2
```

```
False
```

In [8]:
```
1 /= 2
```

```
True
```

In [9]:
```
1 < 2
```

```
True
```

In [10]:
```
1 >= 2
```

```
False
```

## **Argumente funkcij** pišemo brez oklepajev

In [11]:
```
sin pi
```

```
1.2246467991473532e-16
```

In [12]:
```
sin 2 * pi
```

```
2.856642116043664
```

In [13]:
```
sin (2 * pi)
```

```
-2.4492935982947064e-16
```

In [14]:
```
sin 3 + log 10 - min 2 6
```

```
0.44370510105391325
```

## Funkcije kličemo **infiksno** ali **prefiksno**

In [15]:
```
1 + 2
```

```
3
```

In [16]:
```
(+) 1 2
```

```
3
```

In [17]:
```
mod 12345 678
```

```
141
```

In [18]:
```
12345 `mod` 678
```

```
141
```

## Tudi infiksne funkcije lahko **delno uporabimo**

In [19]:
```
razpolovi = (/ 2)
```

In [20]:
```
razpolovi 3
```

```
1.5
```

In [21]:
```
inverz = (1 /)
```

In [22]:
```
inverz 7
```

```
0.14285714285714285
```

In [23]:
```
zadnjaStevka = (`mod` 10)
```

In [24]:
```
zadnjaStevka 12345
```

```
5
```

## Haskell pozna tudi **nabore** in **sezname**

In [25]:
```
zip [1,2,3] "abc"
```

```
[(1,'a'),(2,'b'),(3,'c')]
```

In [26]:
```
[1,2] : [[3,4,5],[6,7]] ++ [[8..15]]
```

```
[[1,2],[3,4,5],[6,7],[8,9,10,11,12,13,14,15]]
```

In [27]:
```
reverse $ take 5 $ reverse [1..100]
```

```
[96,97,98,99,100]
```

## **Nizi** so seznami znakov

In [28]:
```
'A' : 'B' : ['C','D'] ++ "EFG"
```

```
"ABCDEFG"
```

In [29]:
```
reverse "Perica reže raci rep"
```

```
"per icar e\382er acireP"
```

In [30]:
```
length "disproporcioniranost"
```

```
20
```

In [31]:
```
['a'..'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
```

## Haskell vsebuje tudi **izpeljane sezname**

In [32]:
```
[n^2 | n <- [1..10]]
```

```
[1,4,9,16,25,36,49,64,81,100]
```

In [33]:
```
[n | n <- [1..50], n `mod` 7 == 0]
```

```
[7,14,21,28,35,42,49]
```

In [34]:
```
[m * n | m <- [1,2,3], n <- [10,100]]
```

```
[10,100,20,200,30,300]
```

In [35]:
```
[z | z <- "lokomotiva", z /= 'o']
```

```
"lkmtiva"
```

## Funkcije lahko definiramo **po kosih**

In [36]:
```
map f [] = []
map f (x : xs) = f x : map f xs
```

**Use map**

**Found:**

```
map f [] = []
map f (x : xs) = f x : map f xs
```

**Why Not:**

```
map f xs = map f xs
```

In [37]:
```
map (* 5) [1..10]
```

```
[5,10,15,20,25,30,35,40,45,50]
```

## Na definicije lahko dodamo **stranske pogoje**

In [38]:
```
filter p [] = []
filter p (x : xs)
    | p x = x : filter p xs
    | otherwise = filter p xs
```

In [39]:
```
otherwise
```

```
True
```

In [40]:
```
filter even [1..100]
```

```
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,
46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,
88,90,92,94,96,98,100]
```

In [41]:
```
filter ((== 2) . (`mod` 7)) [1..100]
```

```
[2,9,16,23,30,37,44,51,58,65,72,79,86,93,100]
```

## **Pomožne definicije** delamo z `where` in `let`

In [42]:
```
kvadratna a b c
    | d < 0 =
        error "Ni ničel"
    | d == 0 =
        (p, p)
    | otherwise =
        let q = d ** 0.5 / (2 * a) in
        (p - q, p + q)
    where
    d = b ** 2 - 4 * a * c
    p = -b / (2 * a)
```

**Use sqrt**

**Found:** **Why Not:**

`d ** 0.5sqrt d`

# Haskell je **len** jezik

In [43]:
```
ignoriraj x = 0
divergiraj () = divergiraj ()
```

In [44]:
```
ignoriraj (divergiraj ())
```

```
0
```

In [45]:
```
take 10 [1,2..]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

In [46]:
```
take 11 $ "tro" ++ cycle "lo"
```

```
"trolololololo"
```

## Vsak program v Haskellu **ima tip**

In [47]:
```
:t 3 < 5
```

```
3 < 5 :: Bool
```

In [48]:
```
:t [True, False, True]
```

```
[True, False, True] :: [Bool]
```

In [49]:
```
:t ("X", True)
```

```
("X", True) :: (String, Bool)
```

In [50]:
```
:t (||)
```

```
(||) :: Bool -> Bool -> Bool
```

Haskell prav tako podpira **parametrični polimorfizem**

In [51]:
```
:t (++)
```

```
(++) :: forall a. [a] -> [a] -> [a]
```

In [52]:
```
:t repeat
```

```
repeat :: forall a. a -> [a]
```

In [53]:
```
:t []
```

```
[] :: forall a. [a]
```

In [54]:
```
:t zip
```

```
zip :: forall a b. [a] -> [b] -> [(a, b)]
```

## S `type` definiramo **okrajšave**, z `data` **naštevne tipe**

In [55]:
```
type Naslov = String
type Telefon = String
```

In [56]:
```
data Dostava = OsebniPrevzem
             | PoPosti Naslov
             | HitraDostava Naslov Telefon

cena :: Dostava -> Float
cena OsebniPrevzem = 0
cena (PoPosti _) = 2.5
cena (HitraDostava _ _) = 4
```

In [57]:
```
data Maybe a = Nothing
             | Just a
```

# Razredi tipov

Ad-hoc polimorfne funkcije zahtevajo, da tipi pripadajo **razredom**

In [58]:
```
:t max
```

```
max :: forall a. Ord a => a -> a -> a
```

In [59]:
```
:t (==)
```

```
(==) :: forall a. Eq a => a -> a -> Bool
```

In [60]:
```
:t (+)
```

```
(+) :: forall a. Num a => a -> a -> a
```

## Če želimo, lahko Haskell sam **izpelje razrede**

In [61]:
```
data Dostava = OsebniPrevzem
             | PoPosti Naslov
             | HitraDostava Naslov Telefon
             deriving (Eq, Show)
```

In [62]:
```
HitraDostava "Jadranska ulica 21" "01 4766 668"
```

```
HitraDostava "Jadranska ulica 21" "01 4766 668"
```

In [63]:
```
OsebniPrevzem /= PoPosti "Večna pot 113"
```

```
True
```

## Definiramo lahko svoje **razrede** in njihove **pripadnike**

In [64]:
```
class Sized a where
    size :: a -> Integer
```

In [65]:
```
bytes x = (size x + 7) `div` 8
```

In [66]:
```
instance Sized Bool where
    size _ = 1

instance Sized Char where
    size _ = 8
```

In [67]:
```
bytes 'a'
```

```
1
```

## V signaturi lahko naštejemo **več vrednosti**

```
class  Eq a  where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

In [68]:

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

    x /= y = not (x == y)
```

```
<interactive>:1:11: error: [GHC-59692]
    Duplicate instance declarations:
      instance Eq Bool -- Defined at <interactive>:1:11
      instance Eq Bool -- Defined in ‘GHC.Classes’
```

# V razredu lahko tudi **definiramo** vrednosti

```
class  Eq a  where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x /= y  = not (x == y)
    x == y  = not (x /= y)
```

In [ ]:

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

```
<interactive>:1:11: error: [GHC-59692]
    Duplicate instance declarations:
      instance Eq Bool -- Defined at <interactive>:1:11
      instance Eq Bool -- Defined in ‘GHC.Classes’
```

Razredi lahko za svoje člane zahtevajo **dodatne razrede**

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

    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT
    ...
```

Pogoje lahko postavimo tudi pri **pripadnikih**

```
instance  Ord a => Ord [a]  where
  []     <= _      = True
  (_:_)  <= []     = False
  (x:xs) <= (y:ys) = x < y || x == y && xs <= ys
```

In [ ]:

```
instance Sized a => Sized [a] where
    size [] = 0
    size (x : xs) = size x + size xs

instance (Sized a, Sized b) => Sized (a,b) where
    size (x, y) = size x + size y
```

Tipi razreda `Num` podpirajo aritmetiko.

```
class  Num a  where
    (+), (-), (*) :: a -> a -> a
    negate        :: a -> a
    abs           :: a -> a
    signum        :: a -> a
    fromInteger   :: Integer -> a

    x - y         = x + negate y
    negate x      = 0 - x
```

prihodnjič...

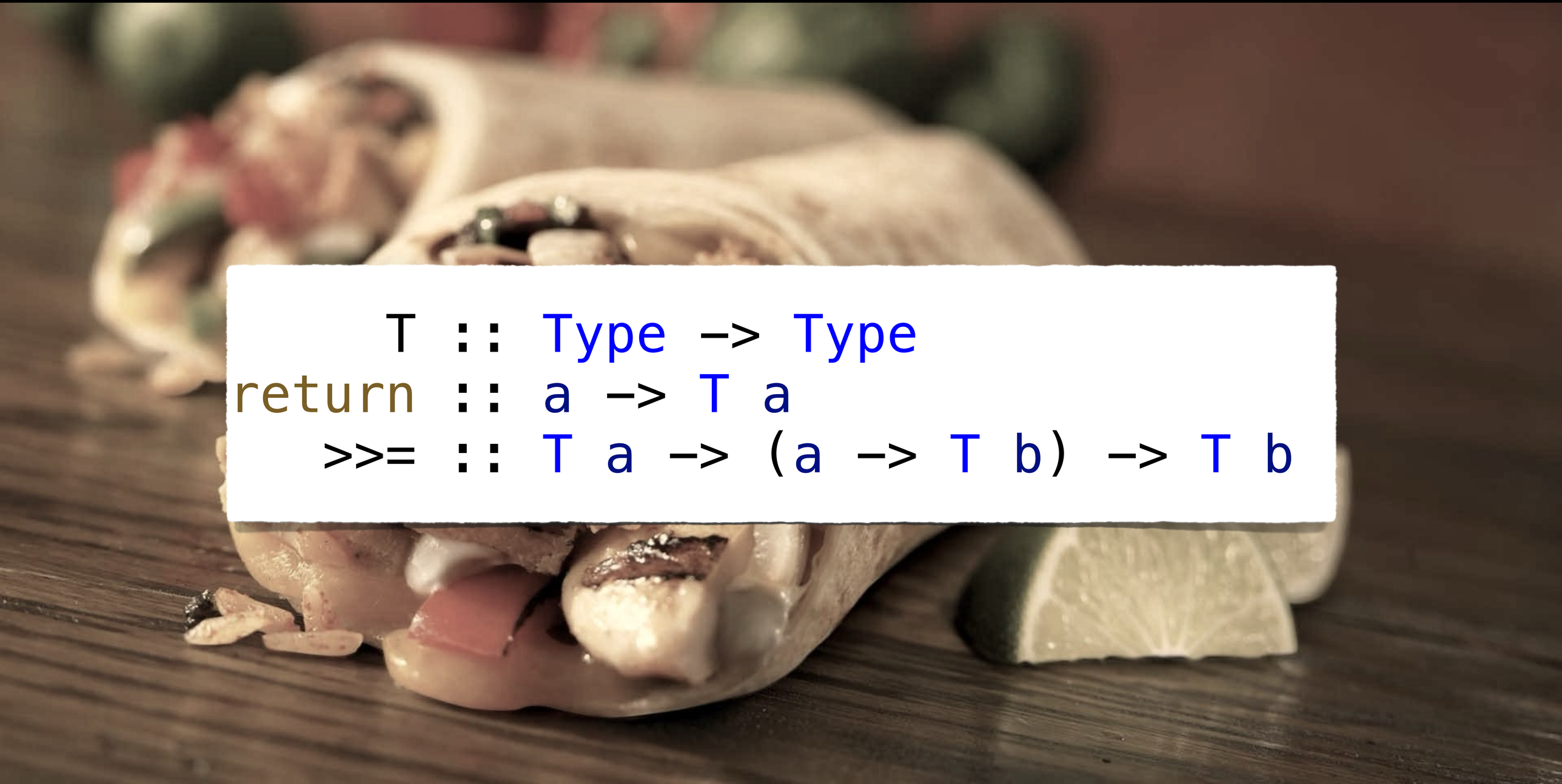
# Posvetili se bomo **računskim učinkom**

program

=

funkcija + **učinki**

# Spoznali bomo **monade**



```
     T :: Type -> Type
return :: a -> T a
   >>= :: T a -> (a -> T b) -> T b
```