Funkcijsko vs. objektno programiranje

drop 3 [1, 2, 3, 4, 5] vs. [1, 2, 3, 4, 5].drop(3)

method receiver a b c vs. receiver.method(a,b,c)

OO programiranje lakše „teče“ – metode objekta navode na način rešavanja

vs.

U funkcijskom programiranju programer preuzima inicijativu

Svi argumenti su jednako važni

Funkcije

U matematici, funkcije su relacije koje imaju **jedinstven** izlaz za dati ulaz.

U matematici

$$zbir : \mathbb{Z} \to \mathbb{Z}$$

$$zbir(a, b) = a + b$$

U Haskell-u

```
zbir :: Int -> Int
zbir a b = a + b
```

Pisanje tipa funkcije nije obavezno, ali je dobra preksa

a i *b* predstavljaju **argumente** funkcije *zbir*.
Prva linija u oba primera opisuje samu funkciju, njen **tip** ili **potpis**.

Matematika vs. funkcijsko programiranje

$f(a, b) + c\ d$ vs. `f a b + c * d`

- Primeniti funkciju f na argumente a i b i dodati proizvod $c\ d$
- Najčešće korišćen simbol zauzima najmanje mesta -- proizvod vs. aplikacija
- Prioritet operacija -- proizvod vs. aplikacija

```
ghci> let f x = x + 3
ghci> f 5 * 7
56
```

Elegantna sintaksa!

Math	Haskell
f(x)	`f x`
f(x,y)	`f x y`
f(g(x))	`f (g x)`
f(x, g(y))	`f x (g x)`
f(x)g(x)	`f x * g x`

Zagrade mogu dodatno da se izbegnu
korišćenjem $ za definisanje desne asocijativnosti

Aplikacija

Sintaksa:

```
funkcija argument1 argument2
```

- Ekvivalent ovog operatora u drugim jezicima bi bio operator poziva funkcije:

```
funkcija(argument1, argument2)
```

Operator aplikacije ima **najviši prioritet**

`negate a + b` je `(negate a) + b`,

za promenu prirориteta koristiti zagrade `negate (a + b)`

Operator aplikacije ima **levu asocijativnost**

`zbir 5 zbir 1 3` je `(zbir 5 zbir) 1 3`

Drugačiji efekat postižemo zagradama `zbir 5 (zbir 1 3)`

Izgled

```
a = 5
b = 10
c = 30
```

```
a = 5
 b = 10
c = 30
```

```
 a = 5
b = 10
  c = 30
```

Izbegava se pisanje dodatnih simbola za ograničavanja blokova

Skript fajl

```
double x = 2 * x
quadruple x = double (double x)
```

```
double x = 2 * x
quadruple = double . double
```

```
ghci> :load Primeri01_1.hs
[1 of 1] Compiling Main             ( Primeri01_1.hs, interpreted )
Ok, one module loaded.
ghci> quadruple 2
8
ghci> :type quadruple
quadruple :: Integer -> Integer
ghci> take (double 3) [1..15]
[1,2,3,4,5,6]
```

Kako u C++/C# izgleda definicija i testiranje ovakvog koda?

Glavni delovi koda i „šum“

Skript fajl

```
faktor n = product [1 .. n]
```

- U proceduralnim jezicima petlja ili rekurzija
- 1*2*…*n
- [1 .. n]
- [1 ..]
- take n [1 ..]

```
faktor n = product (take n [1 .. ])
```

```
average ns = sum ns `div` length ns
```

- Korišćenje funkcija kao infiksnih operatora

```
average ns = div (sum ns) (length ns)
```

Imenovanje

Nazivi funkcija i argumenata počinju malim slovom

Imena tipova počinju velikim slovom

Konvencija

- n – broj
- xs – lista
- xss – lista lista

Kao i u drugim jezicima, rezervisane reči ne mogu da se koriste kao nazivi funkcije:

- `case, class, data, default, deriving, do, else, if, import, in, infix, infixl, infixr, instance, let, module, newtype, of, then, type, where`

Tipovi i klase

Tipovi

Tip je ime za kolekciju povezanih vrednosti

Osnovni tipovi

- `Bool = False | True`

```
Prelude> 1 + False

<interactive>:16:1: error:
    * No instance for (Num Bool) arising from a use of `+'
    * In the expression: 1 + False
      In an equation for `it': it = 1 + False
```

- `Char = 'a' | 'b' | ... | 'A' | 'B' | ...`
- `String`
- `Int = -2^31 | ... | -1 | 0 | 1 | ... | 2^31-1`
- `Integer` (neograničen tip, proizvoljno velike vrednosti)
- `Float`
- `Double`

Logički operatori i funkcije

Logički operatori:

- `&&` (AND)
- `||` (OR)

Logičke funkcije:

- `not :: Bool -> Bool`
- `and :: [Bool] -> Bool`
- `or  :: [Bool] -> Bool`
- `any :: (a -> Bool) -> [a] -> Bool`
- `all :: (a -> Bool) -> [a] -> Bool`

```
xor :: Bool -> Bool -> Bool
xor a b = (a || b) && not (a && b)

-- Alternativni zapis:
-- (a || b) && (not a || not b)
```

Tipovi

```
exp :: Type
```

```
ghci> [1,2,3] :: [Int]
[1,2,3]
ghci> :type [1,2,3]
[1,2,3] :: Num a => [a]
ghci> :type [['a'],['b','c']]
[['a'],['b','c']] :: [[Char]]
ghci> :type ('a', True, 1)
('a', True, 1) :: Num c => (Char, Bool, c)
ghci> :type ('a',(False, "abc"))
('a',(False, "abc")) :: (Char, (Bool, String))
ghci>  :type [1, 2.2, 3]
[1, 2.2, 3] :: Fractional a => [a]
```

Definisanje tipova nije obavezno

Može donekle da ubrza kod

Tipovi lista

Lista je niz elemenata istog tipa

- `[False, True, False] :: [Bool]`
- `['a', 'b', 'c', 'd'] :: [Char]`

`[t]` je lista elemenata tipa `t`

Ista notacija za konstruktor tipa i konstruktor vrednosti

Tip liste ništa ne govori o dužini liste

`[[[Char]]] ?`

Tipovi torki

Torka je niz vrednosti različitog tipa

- `(False, True) :: (Bool, Bool)`
- `(False, 'a', True) :: (Bool, Char, Bool)`
- `(1, True, 'a') :: (Int, Bool, Char)`

`(t1, t2, …, tn)` je n-torka čija i-ta komponenta ima tip `ti` za svako i u [1..n]

Ista notacija za konstruktor tipa i konstruktor vrednosti

Definicija tipa defineše i dužinu/veličinu

Ne postoji ograničenje za tip komponente

- `('a', (False, 'b')) :: (Char, (Bool,Char))`
- `(True, ['a', 'b']) :: (Bool, [Char])`

Tipovi funkcija

```
fun :: t1 -> t2
```

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

zeroto  :: Int -> [Int]
zeroto n = [0 .. n]
```

```
ghci> :type not
not :: Bool -> Bool
ghci> :type div
div :: Integral a => a -> a -> a
ghci> :type product
product :: (Foldable t, Num a) => t a -> a
```

```
ghci> :type add
add :: (Int, Int) -> Int
ghci> :type zeroto
zeroto :: Int -> [Int]
```

```
ghci> :type add
add :: Num a => (a, a) -> a
```

```
ghci> add (3,4)
7
```

Curry-jeve funkcije

Funkcije koje za prosleđen argument vraćaju funkciju

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

```
ghci> add' 3 4
7
ghci> :type add'
add' :: Int -> Int -> Int
```

```
ghci> :type add' 3 4
add' 3 4 :: Int
ghci> :type add' 3
add' 3 :: Int -> Int
ghci> (add' 3) 4
7
```

```
add :: (Int, Int) -> Int    vs.    add' :: Int -> (Int -> Int)
```

Parcijalna aplikacija!

Curry-jeve funkcije

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

Funkcija `mult` prihvata argument x i vraća funkciju `mult x`

Funkcija `mult x` prihvata argument y i vraća funkciju `mult x y`

Funkcija `mult x y` prihvata argument z i vraća rezultat x*y*z

`->` je desno asocijativna

Aplikacija je levo asocijativna

```
ghci> (((mult 2) 3) 4)
24
ghci> mult 2 3 4
24
```

Parcijalna aplikacija

```
ghci> :type take
take :: Int -> [a] -> [a]
ghci> let takeFive = take 5
ghci> :type takeFive
takeFive :: [a] -> [a]
ghci> takeFive [1..]
[1,2,3,4,5]
```

Polimorfne funkcije

Funkcije čiji tipovi argumenti mogu varirati

```
length :: [a] -> Int
```

a = Bool

a = Int

```
ghci> length [False, True]
2
ghci> length [1, 2, 3, 4]
4
```

```
fst :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

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

Klase tipova

~~`sum :: [a] -> a`~~

```
sum :: Num a => [a] -> a
```

`Num` – numerički tipovi

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

`Enum` – nabrojive liste

- `succ :: Enum a => a -> a`

```
ghci> sum [1,2,3]
6
ghci> sum [1.1,2.2,3.3]
6.6
ghci> sum ['a','b','c']

<interactive>:155:1: error:
    * No instance for (Num Char) arising from a use of `sum'
    * In the expression: sum ['a', 'b', 'c']
      In an equation for `it': it = sum ['a', 'b', 'c']
```

```
ghci> ['R'..'f']
"RSTUVWXYZ[\\]^_`abcdef"
```

Klase tipova

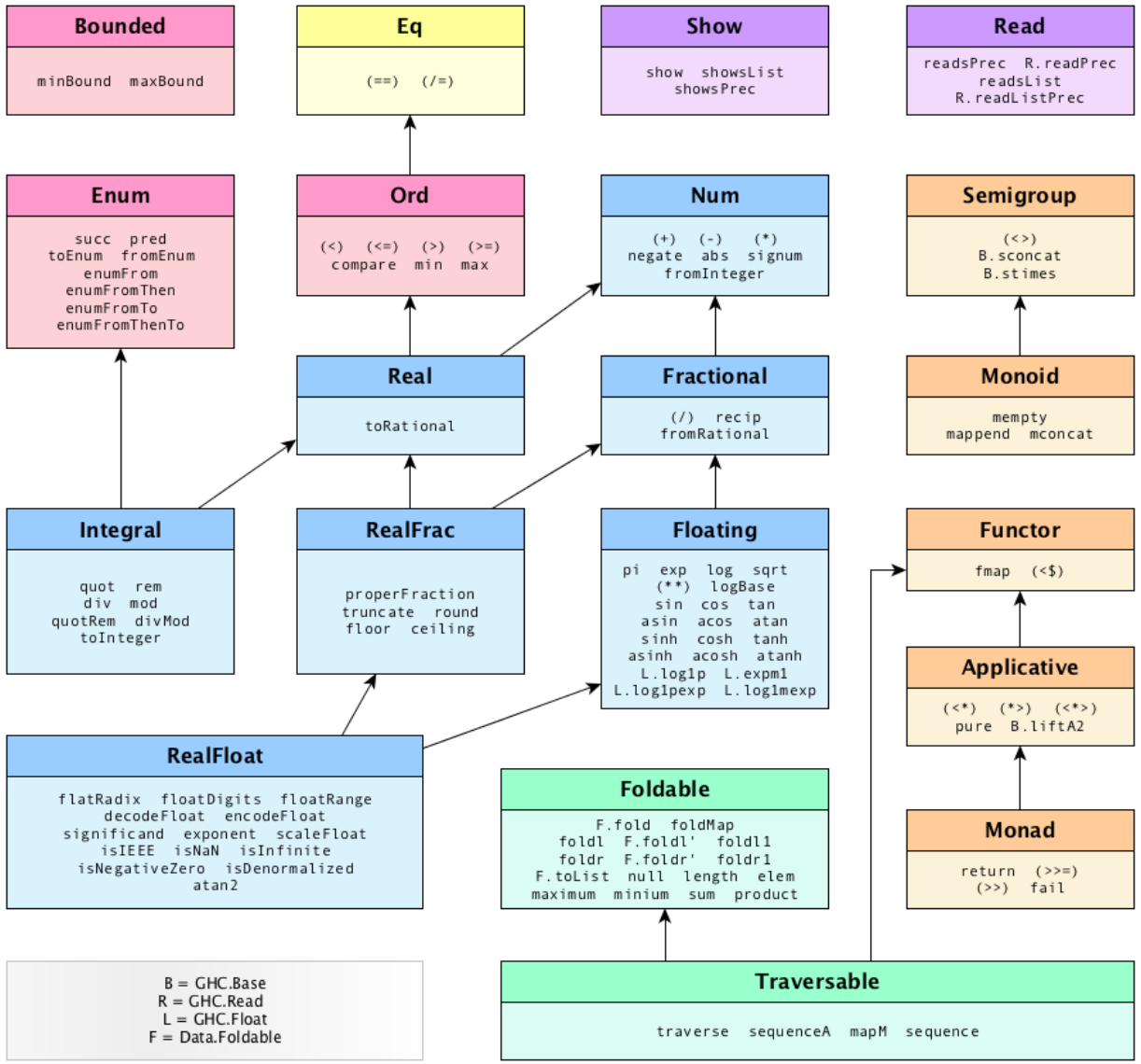
`Eq` – „jednakosni“ tipovi

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

`Ord` – uređeni tipovi

- `10 < 20`
- `'a' < 'b'`
- `"aardvark" < "zzz"`
- `[6,2,4] < [6,3,8]`
- `(<)  :: Ord a => a -> a -> Bool`

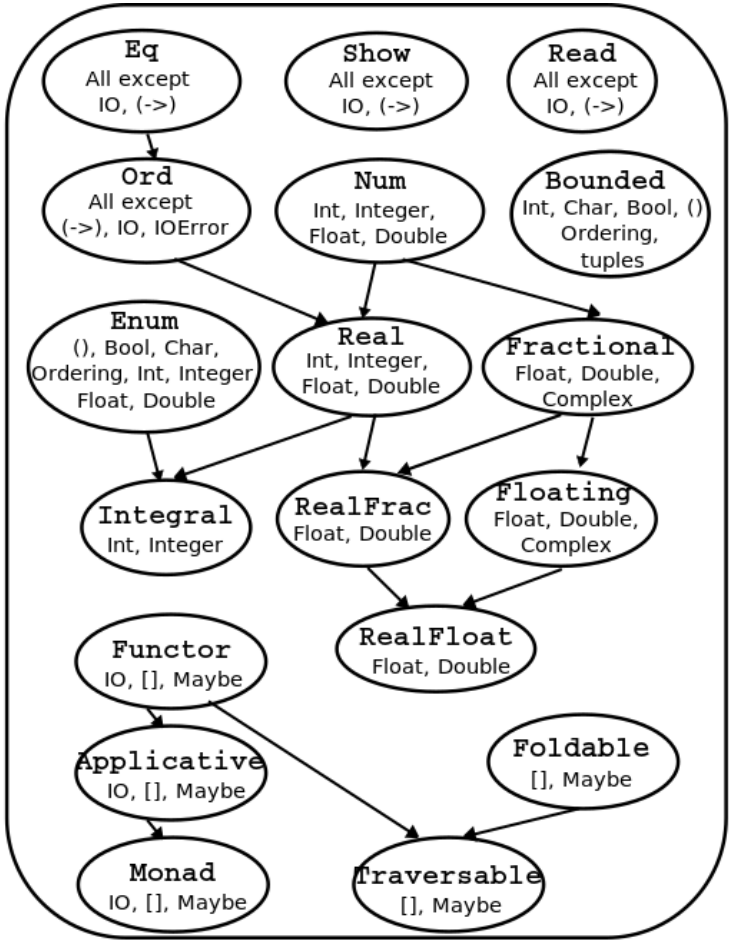
`Show, Read` – tipovi koji se mogu konvertovati u/iz stringa



Hijerahija klasa tipova

Tipovi i klase tipova

Type	Typeclasses
Bool	Eq, Ord, Show, Read, Enum, Bounded
Char	Eq, Ord, Show, Read, Enum, Bounded
Int	Eq, Ord, Show, Read, Enum, Bounded, Num, Real, Integral
Integer	Eq, Ord, Show, Read, Enum, Num, Real, Integral
Float	Eq, Ord, Show, Read, Enum, Num, Real, Fractional, RealFrac, Floating, RealFloat
Double	Eq, Ord, Show, Read, Enum, Num, Real, Fractional, RealFrac, Floating, RealFloat
Word	Eq, Ord, Show, Read, Enum, Bounded, Num, Real, Integral
Ordering	Eq, Ord, Show, Read, Enum, Bounded, Semigroup, Monoid
()	Eq, Ord, Show, Read, Enum, Bounded, Semigroup, Monoid
Maybe a	Eq, Ord, Show, Read, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable
[a]	Eq, Ord, Show, Read, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable
(a,b)	Eq, Ord, Show, Read, Bounded, Semigroup, Monoid, Functor, Applicative, Monad, Foldable, Traversable
a->b	Semigroup, Monoid, Functor, Applicative, Monad
IO	Semigroup, Monoid, Functor, Applicative, Monad
IOError	Eq, Show



Hint and tips

Definisanje funkcije u skript fajlu započeti definisanjem tipa funkcije, iako nije obavezno

- Iz definisanog tipa funkcije vidi se dosta informacija o funkciji

```
add :: Num a => a -> a -> a
add x y = x + y
```

Kada se definišu polimorfne funkcije obratiti pažnju na uključivanje klasa `Num`, `Eq` i `Ord`

Definisanje funkcija

Uslovni izrazi

U većini programskih jezika funkcije se definišu korišćenjem uslovnih izraza

```
abs :: Int -> Int
abs n = if n >= 0 then n else –n
```

Definicija tipa je razdvojena od definicije tela funkcije

vs. `int abs(int n) {…}`

U Haskell-u uslovni izrazi uvek moraju da imaju `else` granu, čime se izbegava moguća dvosmislenost kod ugnježdenih uslova (nema `elif` konstrukcije)

```
signum n = if n<0 then -1 else
                  if n==0 then 0 else 1
```

Guarded equations

G*uarded equations* – cilj je da što više podsećaju na matematičke formule

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

čuvar

Uslov je prebačen na levu stranu jednakosti

Čitljiviji kod kod višestrukih uslova odvajanjem uslova od vrednosti

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

$$sgn(x) = \begin{cases} -1 & , n < 0 \\ 0 & , n = 0 \\ 1 & , inače \end{cases}$$

`otherwise` je definisano kao `True` (ne mora se koristiti samo da u okviru guard-a)

Guards - osnovna sintaksa

```
f x
    | predicate1 = expression 1
    | predicate2 = expression 2
    | predicate3 = expression 3
    | predicate4 = expression 4
```

```
f x
    | predicate1  = expression 1
    | predicate2  = expression 2
    | ...
    | predicate n = expression n
    | otherwise   = expression n+1
```

Isprobajte:

- > otherwise
- > :type otherwise

Guards – primer

```
max :: Int -> Int -> Int

max a b
    | a >= b    = a
    | otherwise = b


maxTri :: Int -> Int -> Int -> Int
max a b c
    | a >= b && a >= c = a
    | b >= c           = b
    | otherwise        = c
```

```
max 4 12


maxTri 5 11 8
```

Možemo zapisivati i kompleksnije izraze

Layout

```
----- v1 -----
xy n | n == 0 = 1
     | n == 1 = 2
     | n == 2 = 3
     | n == 3 = 4
```

```
----- v2 -----
xy n | n == 0 = 1
    | n == 1 = 2      -- tab
    | n == 2 = 3      -- tab
       | n == 3 = 4   -- tab
```

```
----- v3 -----
xy n | n == 0 = 1
    | n == 1 = 2          -- tab
       | n == 2 = 3       -- 2 x tab
          | n == 3 = 4    -- 2 x tab
```

```
----- v4 -----
xy n | n == 0 = 1
    | n == 1 = 2      -- tab
       | n == 2 = 3   -- 2 x tab
       | n == 3 = 4   -- tab
```

```
----- v5 -----
xy n | n == 0 = 1
    | n == 1 = 2      -- tab
       | n == 2 = 3   -- 2x tab
     | n == 3 = 4
```

```
----- v6 -----
xy n | n == 0 = 1
    | n == 1 = 2      -- tab
       | n == 2 = 3   -- 2 x tab
     | n == 3 = 4
```

```
----- v7 -----
xy n | n == 0 = 1
    | n == 1 = 2      -- tab
       | n == 2 = 3   -- 2x tab
 | n == 3 = 4
```

```
----- v8 -----
xy n
  | n == 0 = 1
  | n == 1 = 2
  | n == 2 = 3
  | n == 3 = 4
```

Layout

Iako prethodni primeri rade, većina njih se teško čita i tumači, tako da to nije nešto što treba primenjivati. Preporučuje se korišćenje nekog od sledećih formata.

```
----- v1 -----
xy n | n == 0 = 1
     | n == 1 = 2
     | n == 2 = 3
     | n == 3 = 4
```

```
----- v8 -----
xy n
   | n == 0 = 1
   | n == 1 = 2
   | n == 2 = 3
   | n == 3 = 4
```

Faktorijel

```
fakt :: Int -> Int

fakt n
    | n == 0 = 1
    | n > 0  = n * fakt (n-1)
```

```
fakt 5
```

Isprobati sa i bez definicije tipa fukcije

Vraćanje više od jedne vrednosti

Ukoliko želimo da funkcija vrati više od jednog izlaza, možemo koristiti torke.

```
minAndMax :: Int -> Int -> (Int, Int)
minAndMax a b
      | a >= b    = (b, a)
      | otherwise = (a, b)
```

Pattern matching

Za mnoge funkcije čitljiviji način definisanje

```
not   :: Bool -> Bool
not False = True
not True  = False
```

Ovakav pristupa omogućava i definiciju funkcija sa više argumenata

Pattern matching

```
(&&) :: Bool -> Bool -> Bool
True && True   = True
True && False  = False
False && True  = False
False && False = False
```

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

```
(&&) :: Bool -> Bool -> Bool
True && x  | x == True   = True
           | x == False  = False
False && x | x == True   = False
           | x == False  = False
```

```
(&&) :: Bool -> Bool -> Bool
True && b = b
False && _ = False
```

Ne mora da bude
izračunato pre primene

Pattern matching

```
(&&) :: Bool -> Bool -> Bool
True && b = b
False && _ = False
```

Tip `Bool` ima TRI vrednost – `True, False, undetermined (bottom, ⊥)`

- Zbog „lenjosti“ svaki tip ima i vrednost `undefined`

```
ghci> head []
*** Exception: Prelude.head: empty list
ghci> True && head []
*** Exception: Prelude.head: empty list
ghci> False && head []
False
ghci> let f x = 456
ghci> f (True && head [])
456
```

Pattern matching

Zamena redosleda šablona daće drugačiji rezultat

```
(&&) :: Bool -> Bool -> Bool
_    && _    = False
True && True = True
```

→ Vraća uvek `False`

Top-to-bottom, Left-to-right

Nije dozvoljeno koristiti isto ime za više od jednog argumenta

~~
```
(&&) :: Bool -> Bool -> Bool
b && b = b
_ && _ = False
```
~~

```
(&&) :: Bool -> Bool -> Bool
b && c | b == c    = b
       | otherwise = False
```

Šabloni torki

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

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

Šablon liste

```
[1,2,3,4] :: [Int]   ↔  1: (2: (3 : (4 : [])))
```

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

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

Šablon liste

```
head :: [a] -> a
head (x: _) = x
```

`head [] ?`

- Šta kada pošaljemo upit serveru, pa čekamo odgovor?
  - Da li je došlo do greške ili se zahtev još uvek obrađuje?

```
ghci> let f x = 475
ghci> f (head [])
475
```

f ⊥ = 475

Zašto su zagrade neophodne?

Šta bi značilo `head x :_ = x` ?

`(head x) : _ = x`

Errors ⇔ Unterminated computations

Šablon liste

```
tail :: [a] -> [a]
tail (_: xs) = xs
```

Na osnovu tipa funkcije može se dosta saznati o samoj funkciji

```
tail :: a -> a
tail xs = xs

tail :: b -> [a]
tail xs = []

tail :: a -> b
tail xs = ⊥
```

Lambda izrazi

```
ghci> (\x -> x + x) 2
4
```

Izbegavanje imenovanja funkcije koja se samo jednom koristi

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

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

Sekcija operatora

Operatori se mogu koristiti kao funkcije

- `(+) 2 3` ili `2 + 3`

Ako je # operator, tada se izrazi oblika `(#)`, `(x #)` i `(# y)` nazivaju sekcije

- `(#) = \x -> (\y -> x # y)`
- `(x #) = \y -> x # y`
- `(# y) = \x -> x # y`

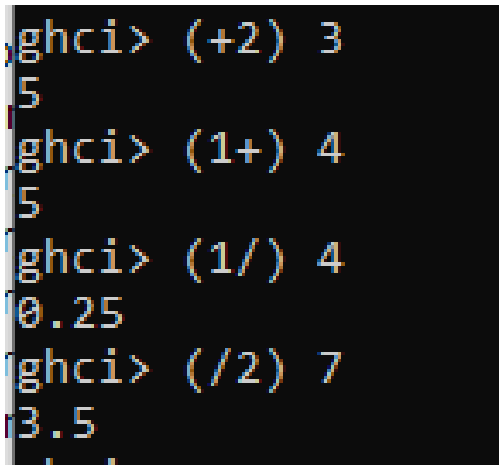
- `(\x -> 1 + x) 3` → `1 + 3`

  višak

- `(1+) 3` → `1 + 3`

Sekcije operatora - primena

1. Definisanje jednostavnih funkcija
   - `(+)` sabiranje -- `\x -> (\y -> x + y)`
   - `(1+)` naslednik -- `\y -> 1 + y`
   - `(1/)` recipročna vrednost -- `\y -> 1 / y`
   - `(*2)` dvostruka vrednost -- `\x -> x * 2`
   - `(/2)` polovina vrednosti -- `\x -> x / 2`



Sekcije operatora - primena

2. Naglašavanje tipa operatora
   - `(+) :: Int -> Int -> Int`

3. Prosleđivanje operatora kao argumenta funkcije
   - ```
     sum :: [Int] -> Int
     sum = foldl (+) 0
     ```
     ```