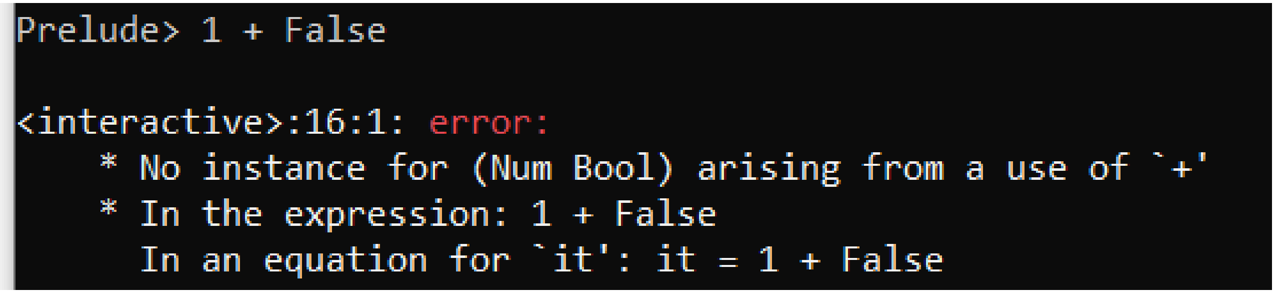
Tipovi i klase

Tipovi

Tip je ime za kolekciju povezanih vrednosti

Osnovni tipovi

- `Bool = False | True`

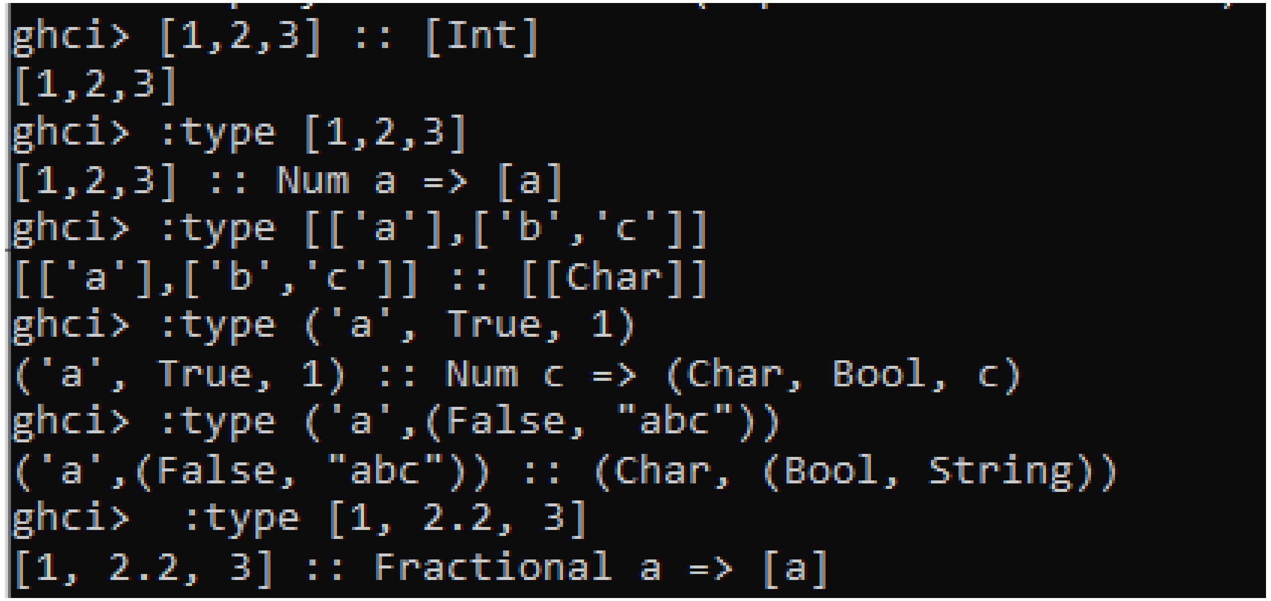


```
Prelude> 1 + False

<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`

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

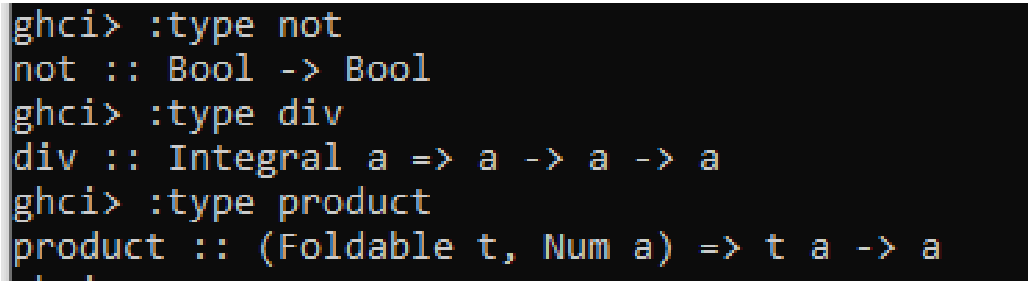
Definicija tipa defineše i dužinu/veličinu

Ne postoji ograničenje za tip komponente

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

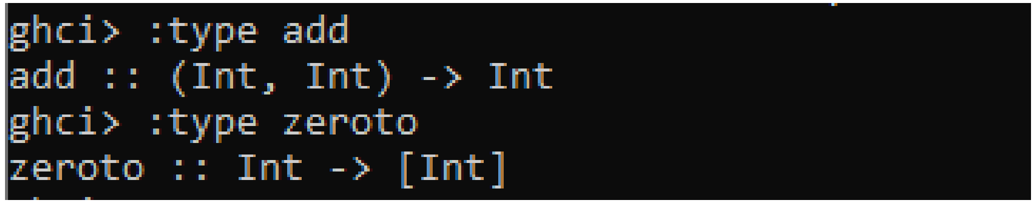
Tipovi funkcija

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

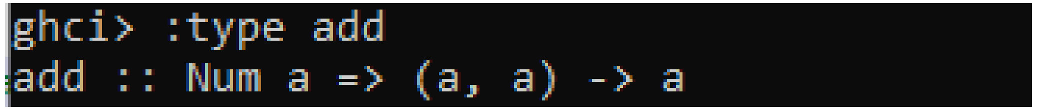


```
ghci> :type not
not :: Bool -> Bool
ghci> :type div
div :: Integral a => a -> a -> a
ghci> :type product
product :: (Foldable t, Num a) => t a -> a
```

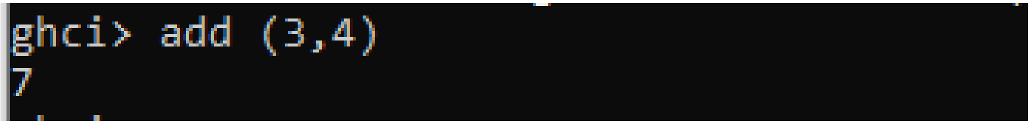
```
~~add   :: (Int, Int) -> Int~~
add (x,y) = x + y
add = \(x,y) -> x + y

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



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



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

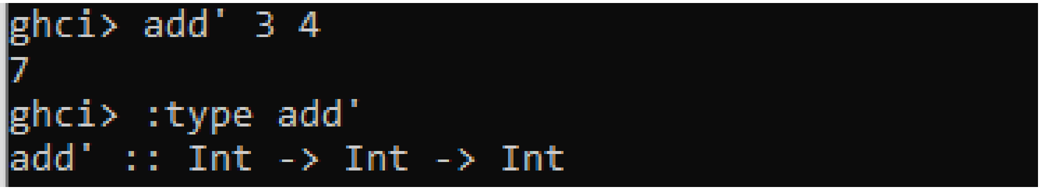


```
ghci> add (3,4)
7
```

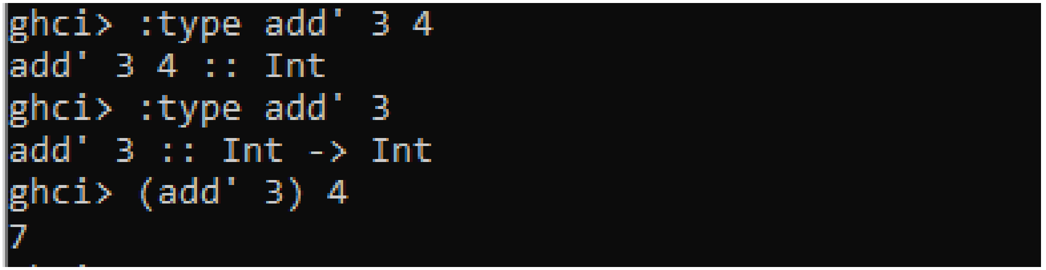
Curry-jeve funkcije

Funkcije koje za prosleđen argument vraćaju funkciju

```haskell
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
```

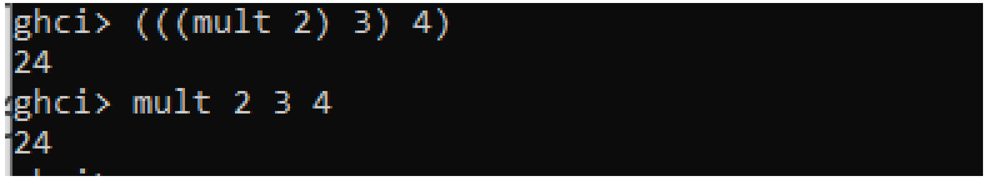
Funkcija `mult` prihvata argument x i vraća funkciju `mult x`

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`

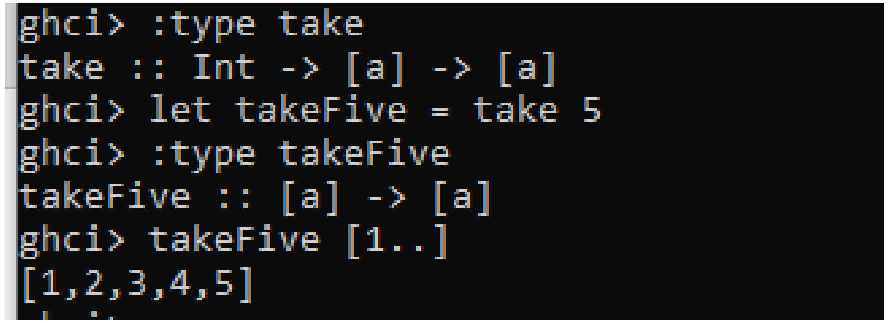
`->` je desno asocijativna

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]
```

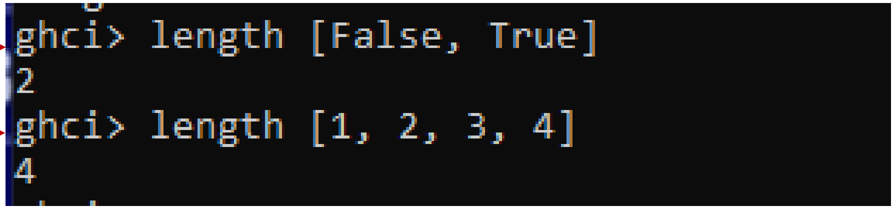
Polimorfne funkcije

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~~

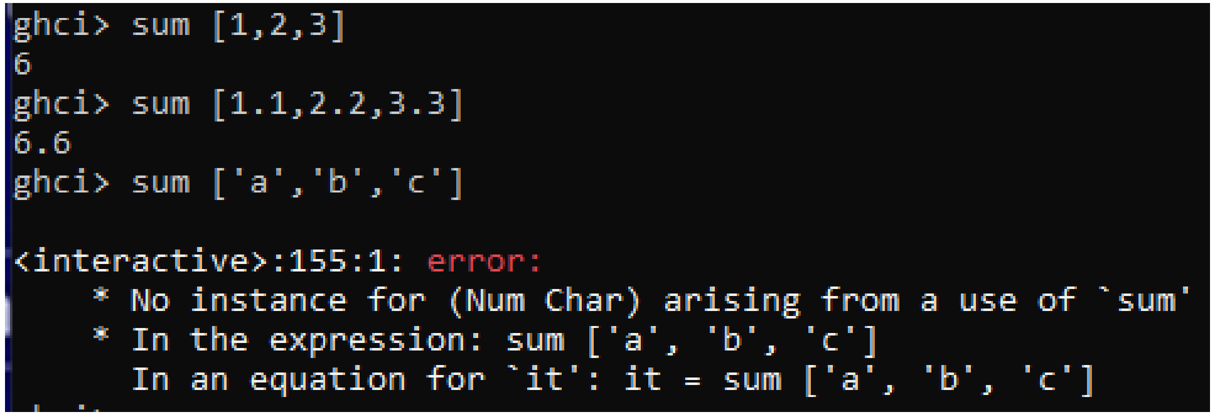
sum :: Num a => [a] -> a

Num – numerički tipovi

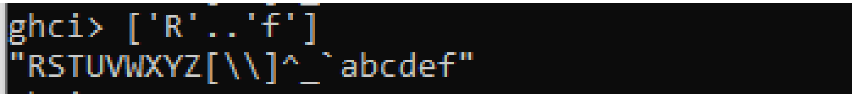
- (+)   :: Num a => a -> a -> a

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

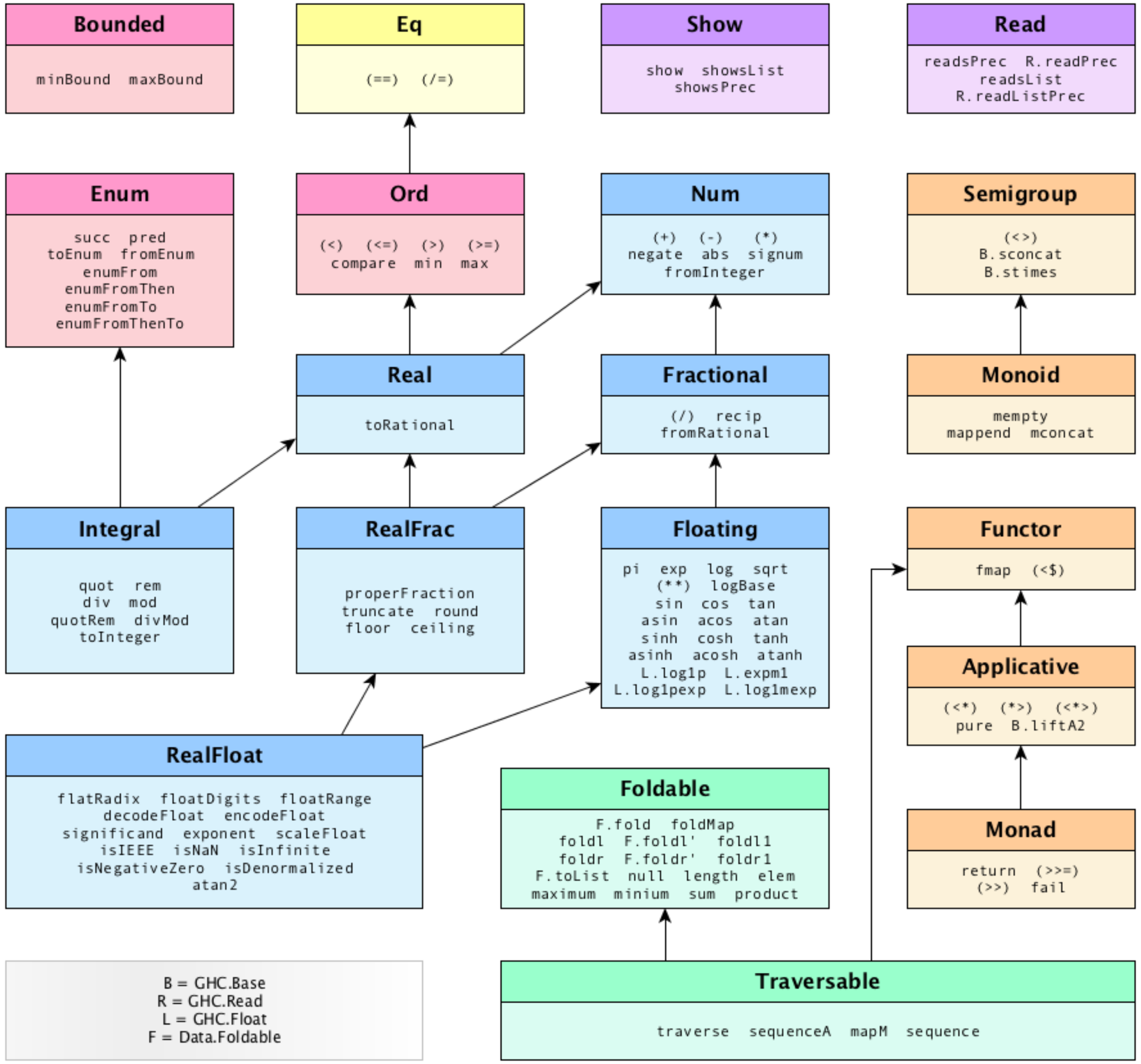
- `(==) :: Eq a => a -> a -> Bool`

`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



Bounded

minBound maxBound

Eq

(==) (/=)

Show

show showsList
showsPrec

Read

readsPrec R.readPrec
readsList
R.readListPrec

Enum

succ pred
toEnum fromEnum
enumFrom
enumFromThen
enumFromTo
enumFromThenTo

Ord

(<) (<=) (>) (>=)
compare min max

Num

(+) (-) (*)
negate abs signum
fromInteger

Semigroup

(<>)
B.sconcat
B.stimes

Real

toRational

Fractional

(/) recip
fromRational

Monoid

mempty
mappend mconcat

Integral

quot rem
div mod
quotRem divMod
toInteger

RealFrac

properFraction
truncate round
floor ceiling

Floating

pi exp log sqrt
(**) logBase
sin cos tan
asin acos atan
sinh cosh tanh
asinh acosh atanh
L.log1p L.expm1
L.log1pexp L.log1mexp

Functor

fmap (<$)

Applicative

(<*) (*>) (<*>)
pure B.liftA2

RealFloat

flatRadix floatDigits floatRange
decodeFloat encodeFloat
significand exponent scaleFloat
isIEEE isNaN isInfinite
isNegativeZero isDenormalized
atan2

Foldable

F.fold foldMap
foldl F.foldl' foldl1
foldr F.foldr' foldr1
F.toList null length elem
maximum minium sum product

Monad

return (>>=)
(>>) fail

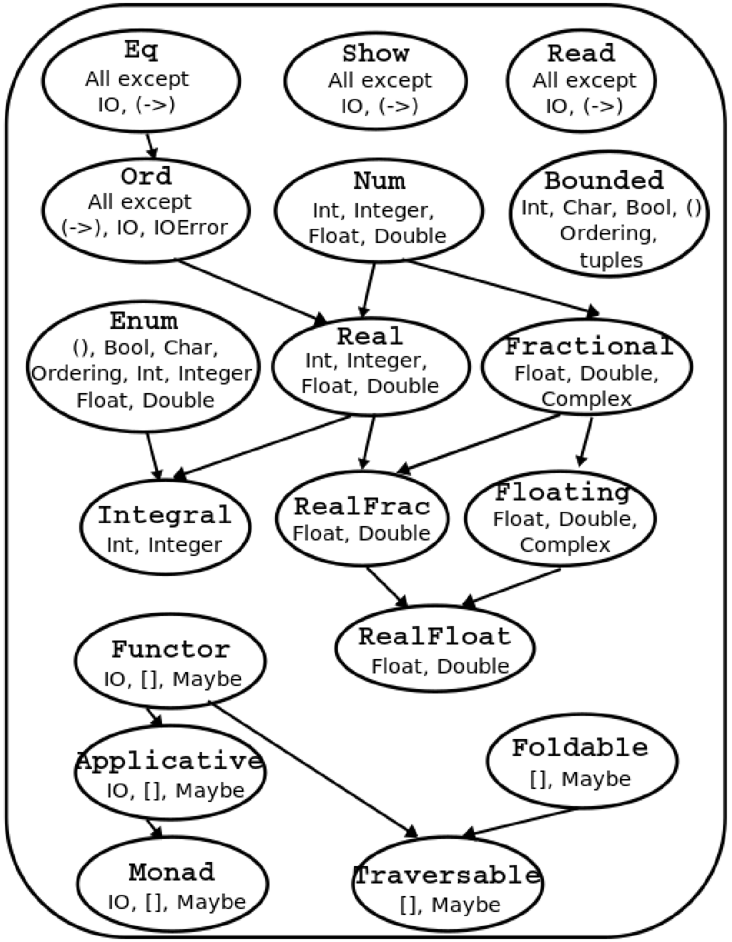
B = GHC.Base
R = GHC.Read
L = GHC.Float
F = Data.Foldable

Traversable

traverse sequenceA mapM sequence

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



Eq All except IO, (->)

Show All except IO, (->)

Read All except IO, (->)

Ord All except (->), IO, IOError

Num Int, Integer, Float, Double

Bounded Int, Char, Bool, () Ordering, tuples

Enum (), Bool, Char, Ordering, Int, Integer Float, Double

Real Int, Integer, Float, Double

Fractional Float, Double, Complex

Integral Int, Integer

RealFrac Float, Double

Floating Float, Double, Complex

RealFloat Float, Double

Functor IO, [], Maybe

Applicative IO, [], Maybe

Foldable [], Maybe

Monad IO, [], Maybe

Traversable [], Maybe

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

Guarded 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`

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 && x  | x == True   = True
           | x == False  = False
False && x | x == True   = False
           | x == False  = False
```

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

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

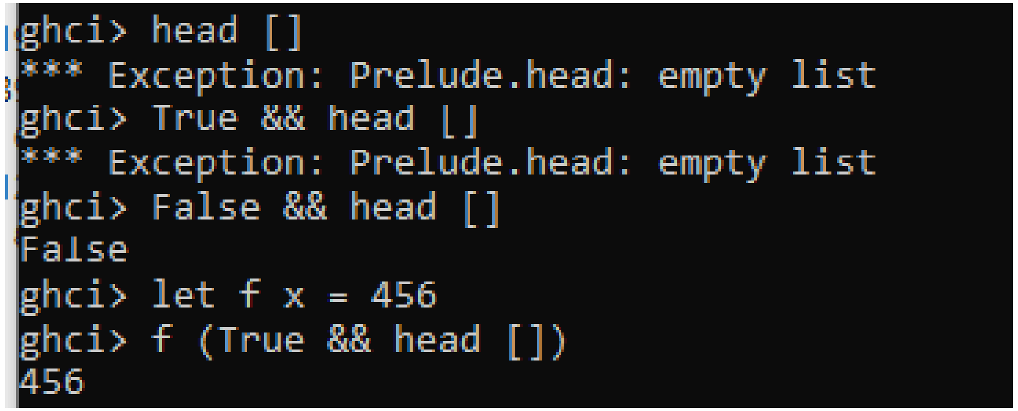
Ne mora da bude
izračunato pre primene

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
```

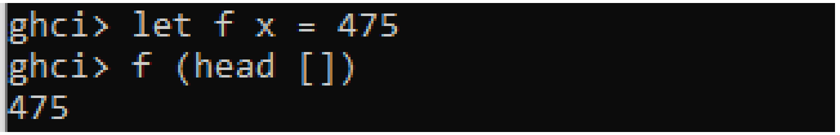
```
test :: [Char] -> Bool
test ('a': _) = True
test _        = False
```

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~~

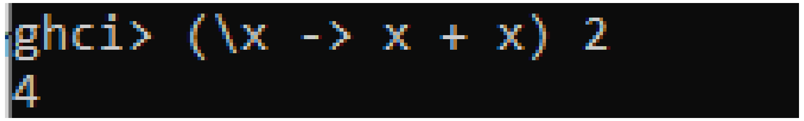
~~tail :: b -> [a]~~

~~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

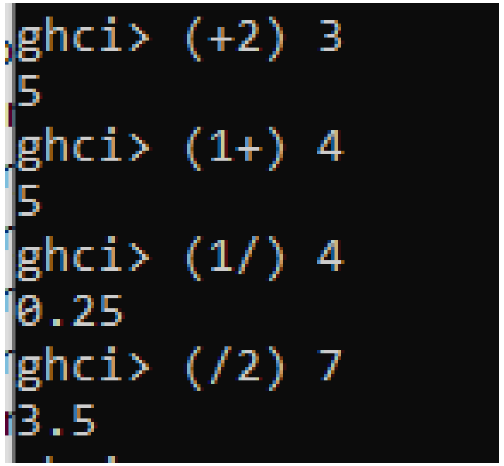
- (\x -> 1 + x) 3 → 1 + 3

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`



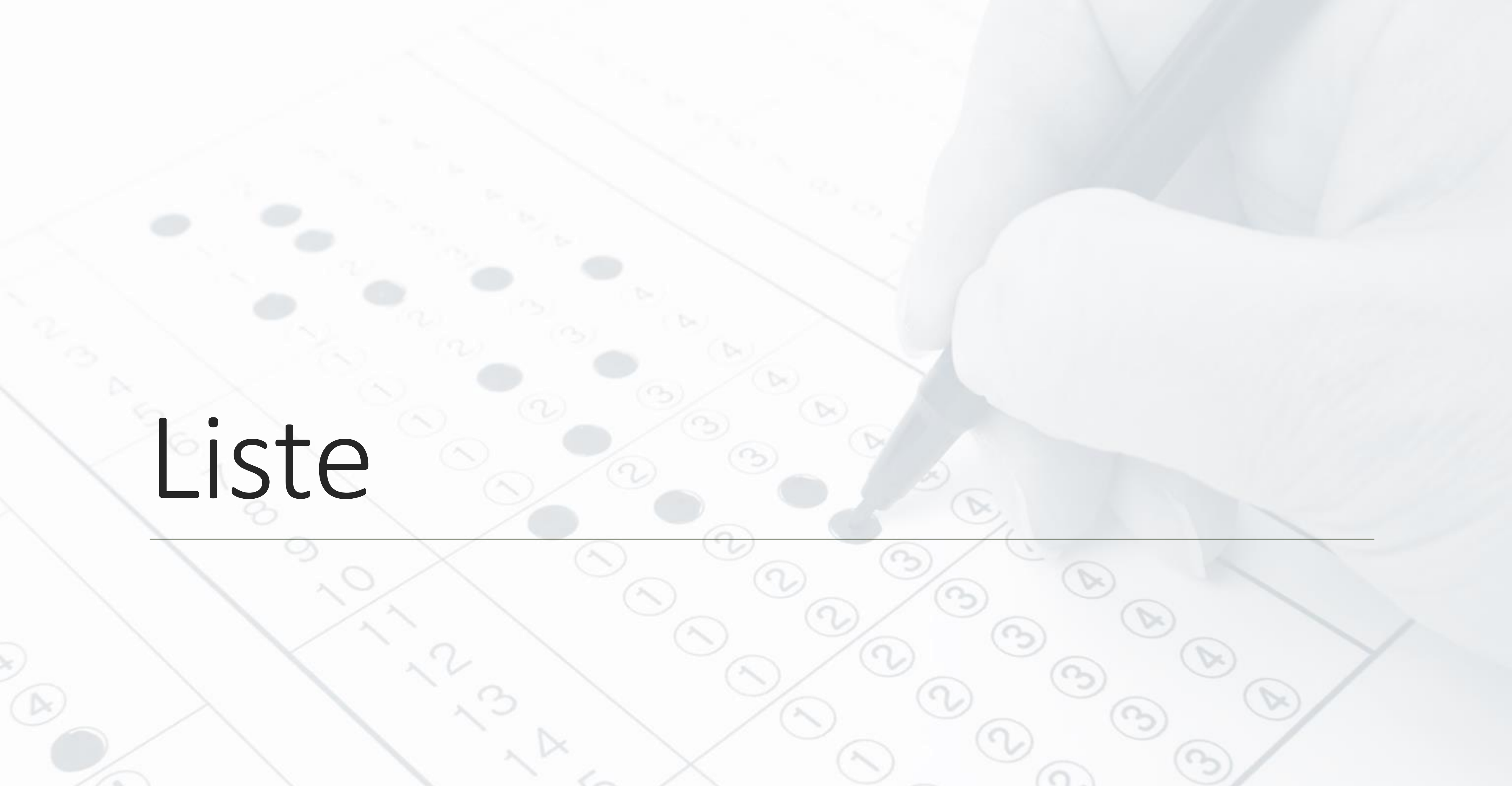
```
ghci> (+2) 3
5
ghci> (1+) 4
5
ghci> (1/) 4
0.25
ghci> (/2) 7
3.5
```

Sekcije operatora - primena

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

3. Prosleđivanje operatora kao argumenta funkcije

   -
     ```
     sum :: [Int] -> Int
     sum = foldl (+) 0
     ```

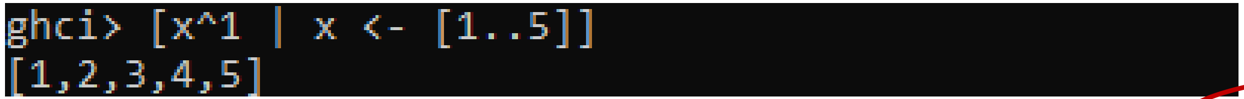


Liste

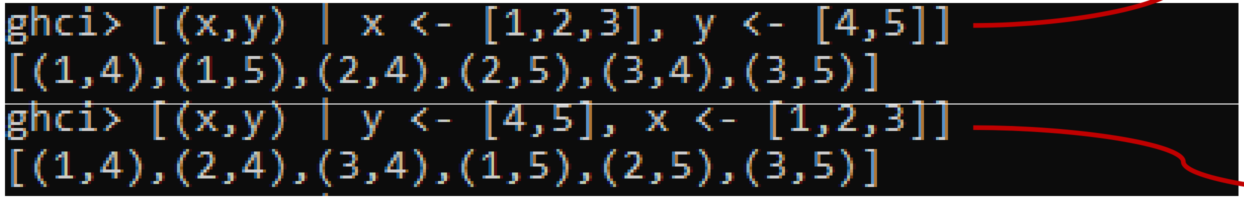
Liste

Liste imaju koren u matematičkoj definiciji skupova

Lists comprehensions



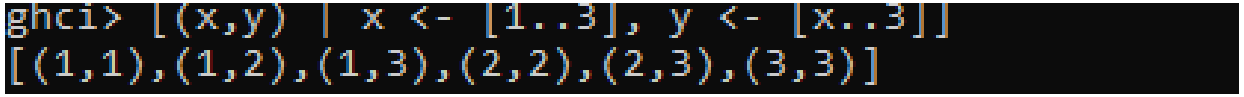
```
ghci> [x^1 | x <- [1..5]]
[1,2,3,4,5]
```



```
ghci> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
ghci> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

```
for (i=1; i<=3; i++)
    for(j=4; j<=5; j++)
          …
```

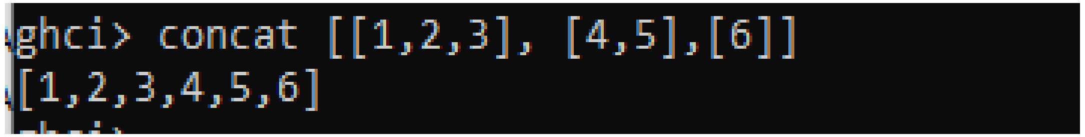
```
for (j=4; j<=5; j++)
    for(j=1; i<=3; i++)
          …
```



```
ghci> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

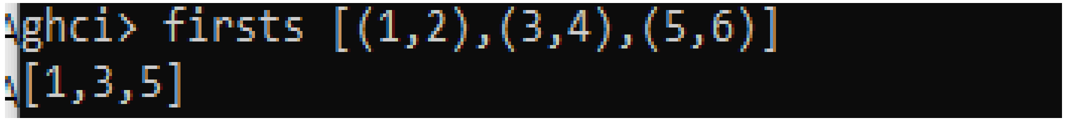
Liste

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```



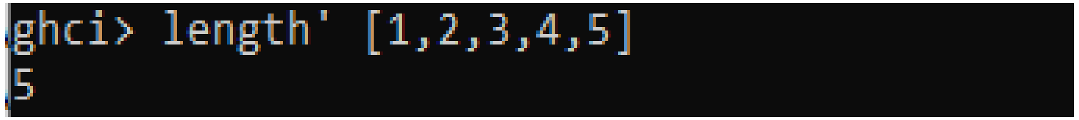
```
ghci> concat [[1,2,3], [4,5],[6]]
[1,2,3,4,5,6]
```

```
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]
```



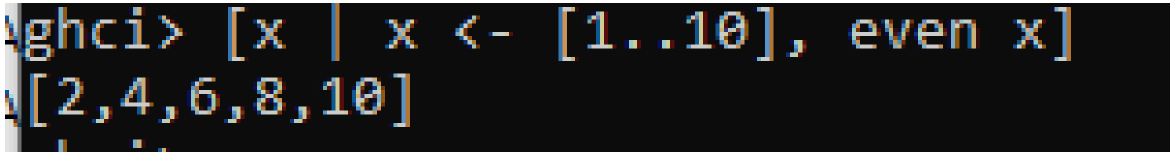
```
ghci> firsts [(1,2),(3,4),(5,6)]
[1,3,5]
```

```
length’ :: [a] -> Int
length’ xs = sum [1 | _ <- xs]
```



```
ghci> length' [1,2,3,4,5]
5
```

„Čuvari“ u listama

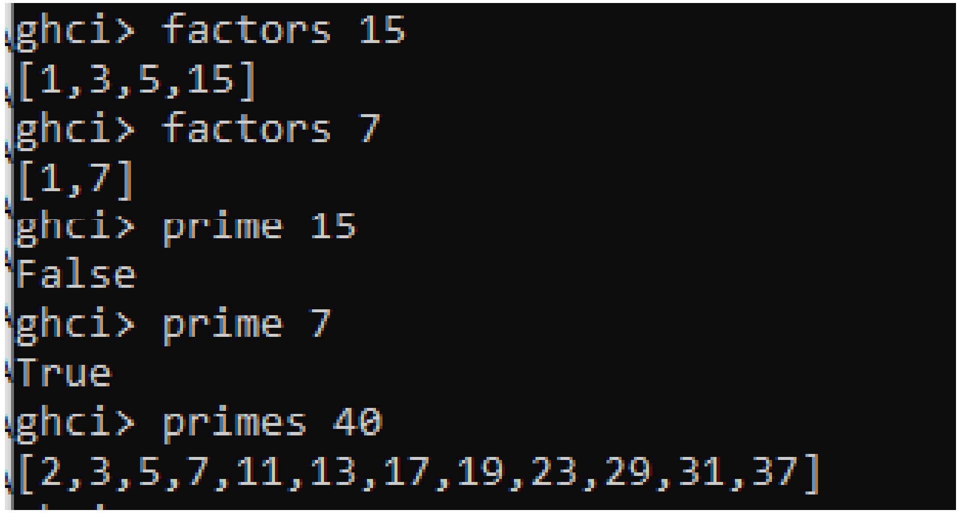


```
ghci> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

```
prime :: Int -> Bool
prime n = factors n == [1,n]
```

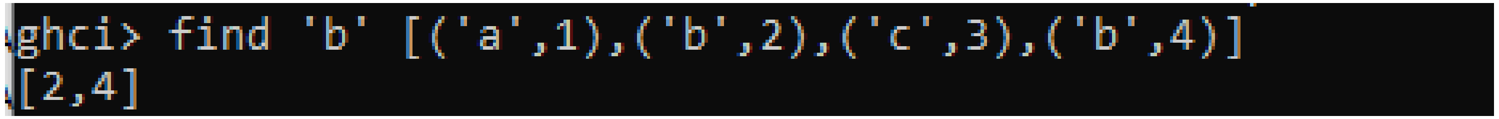
```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```



```
ghci> factors 15
[1,3,5,15]
ghci> factors 7
[1,7]
ghci> prime 15
False
ghci> prime 7
True
ghci> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

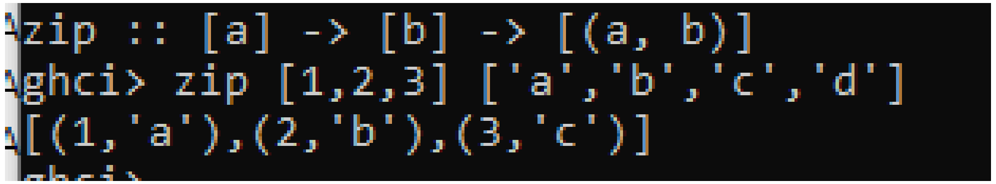
„Čuvari“ u listama

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



```
ghci> find 'b' [('a',1),('b',2),('c',3),('b',4)]
[2,4]
```

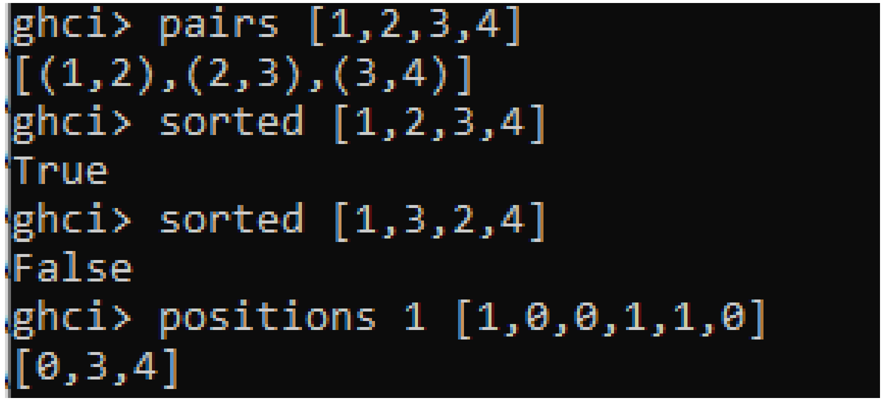
Funkcija zip



```
zip :: [a] -> [b] -> [(a, b)]
ghci> zip [1,2,3] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c')]
ghci>
```

```
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]
```



```
ghci> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
ghci> sorted [1,2,3,4]
True
ghci> sorted [1,3,2,4]
False
ghci> positions 1 [1,0,0,1,1,0]
[0,3,4]
```

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x’,i) <- zip xs [0..], x == x’]
```