Stringovi

String comprehensions

```
"abc" :: String
ghci> "abcde" !! 2
'c'
ghci> take 3 "abcde"
"abc"
ghci> length "abcde"
5
ghci> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Stringovi

Broj malih slova u tekstu

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, x >= ’a’ && x <= ’z’]
```

```
ghci> lowers "Haskell"
6
```

Broj pojavljivanja nekog slova u tekstu

```
count :: Char -> String -> Int
count x xs = length [x’ | x’ <- xs, x == x’]
```

```
ghci> count 's' "Mississippi"
4
```

Primer. Cezarova šifra

Kodiranje nazvano po Juliju Cezaru koji je ovu metodu koristio pre više od 2000 godina

Svako slovo u tekstu se menja slovom koje je udaljeno N pozicija u abcedi

„haskell is fun“ → „kdvnhoo lv ixq„

Cezarova šifra – enkodiranje i dekodiranje

`import Data.Char` – zbog korišćenja funkcija definisanih u ovom modulu

```
let2int :: Char -> Int
let2int c = ord c - ord ’a’

int2let :: Int -> Char
int2let n = chr (ord ’a’ + n)
```

```
ghci> let2int 'a'
0
ghci> int2let 2
'c'
```

Cezarova šifra – enkodiranje i dekodiranje

```
shift :: Int -> Char -> Char
shift n c | isLower c = int2let ((let2int c + n) `mod` 26)
          | otherwise = c
```

```
ghci> shift 3 'a'
'd'
ghci> shift 3 'z'
'c'
ghci> shift (-3) 'a'
'x'
```

```
encode :: Int -> String -> String
encode n xs = [shift n x | x <- xs]
```

```
ghci> encode 3 "haskell is fun"
"kdvnhoo lv ixq"
ghci> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

Cezarova šifra – tabela frekvencija

Za razbijanje Cezarove šifre neophodno je poznavanje frekvencije određenih slova u jeziku

Analizom velikog broja tekstova na Engleskom jeziku dobijena je tabela

```
table :: [Float]
table = [8.1, 1.5, 2.8, 4.2, 12.7, 2.2, 2.0, 6.1, 7.0,
         0.2, 0.8, 4.0, 2.4, 6.7, 7.5, 1.9, 0.1, 6.0,
         6.3, 9.0, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

Cezarova šifra – tabela frekvencija

```
percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

Obavezna konverzija numeričkih tipova U Num

```
ghci> percent 5 15
33.333336
```

```
freqs :: String -> [Float]
freqs xs = [percent (count x xs) n | x <- ['a'..'z']]
                where n = lowers xs
```

```
ghci> freqs "abbcccddddeeeee"
[6.666667,13.333334,20.0,26.666668,33.333336,0.0,0.0,0.
```

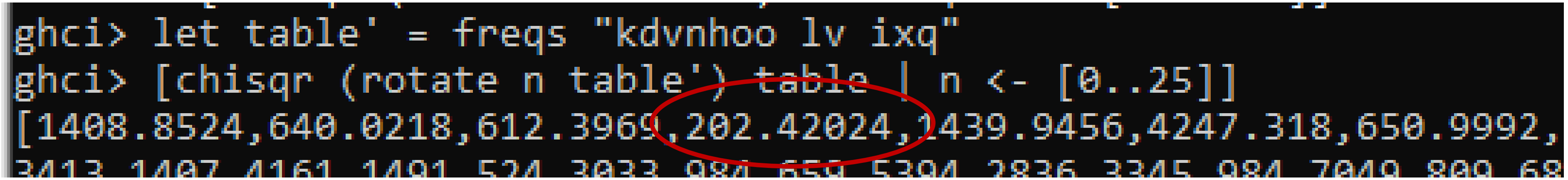
Cezarova šifra – razbijanje šifre

Za razbijanje šifre koristi se metoda poređenja frekvencije pojave slova u teksta sa očekivanim frekvencijama χ^2 statistikom

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

```
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [((o-e)^2)/e | (o,e) <- zip os es]
```

```
rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs
```



Cezarova šifra – razbijanje šifre

```
crack :: String -> String
crack xs = encode (-factor) xs
           where
              factor = head (positions (minimum chitab) chitab)
              chitab = [chisqr (rotate n table') table | n <- [0..25]]
              table' = freqs xs
```

```
ghci> crack "kdvnhoo lv ixq"
"haskell is fun"
ghci> crack "vscd mywzboroxcsyxc kbo ecopev"
"list comprehensions are useful"
ghci> crack (encode 3 "haskell")
"piasmtt"
ghci> crack (encode 3 "boxing wizards jump quickly")
"wjsdib rduvmyn ephk lpdxfgt"
```

Rekurzije

Rekurzivne funkcije

```haskell
fac :: Int -> Int
fac n = product (take n [1..])
```

Kompozicija

```haskell
fac 4
  = product (take 4 [1..])
  = product [1, 2, 3, 4]
  = 1 * 2 * 3 * 4
  = 120
```

```haskell
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Rekurzija

```haskell
fac 3
  = 3 * fac 2
  = 3 * (2 * fac 1)
  = 3 * (2 * (1 * fac 0))
  = 3 * (2 * (1 * 1))
  = 3 * (2 * 1)
  = 3 * 2
  = 6
```

Rekurzivne funkcije

```haskell
fac :: Int -> Int
fac n = product (take n [1..])
```

Kompozicija

```haskell
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Rekurzija

```
ghci> fac -1

<interactive>:2:5: error:
    * No instance for (Num (Int -> Int)) arising from a use of `-'
        (maybe you haven't applied a function to enough arguments?)
    * In the expression: fac - 1
      In an equation for `it': it = fac - 1
```

```
ghci> fac (-1)
1
```

```
ghci> fac' (-1)
Interrupted.
```

Zašto je rekurzija korisna?

Haskell nema petlje/iteraciju!

Jednostavnije definisanje mnogih funkcija

Osobine funkcija definisanih rekurzivno se lako dokazuju matematičkom indukcijom

Rekurzije nad listama

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * product xs
```

```
product [1,2,3] =
  = 1 * product [2,3]
  = 1 * (2 * product [3])
  = 1 * (2 * (3 * product []))
  = 1 * (2 * (3 * 1))
  = 1 * (2 * 3)
  = 1 * 6
  = 6
```

Rekurzije nad listama

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Ista struktura definicije funkcije – moći ćemo to da iskoristimo

- Šta je vrednost prazne liste?
- Šta radimo sa ostatkom liste?

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse [1,2.3]
  = reverse [2,3] ++ [1]
  = (reverse [3] ++ [2]) ++ [1]
  = ((reverse [] ++ [3]) ++ [2]) ++ [1]
  = (([] ++ [3]) ++ [2]) ++ [1]
  = [3,2,1]
```

Rekurzivne funkcije sa više argumenata

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
drop :: Int -> [a] -> [a]
drop 0 xs      = xs
drop _ []      = []
drop n (_:xs) = drop (n-1) xs
```

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       = ys
(x:xs) ++ ys  = x : (xs ++ ys)
```

Višestruka rekurzija

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                   smaller = [a | a <- xs, a <= x]
                   larger  = [b | b <- xs, b > x]
```

Najjednostavnija implementacija Quick sort algoritma

Mana – nije zadržana osnovna ideja Quick sorta, tj. korišćenje istog memorijskog prostora

Zgodno za paralelizaciju!

„Uzajamna“ rekurzija

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

```
even 5 =
   = odd 4
   = even 3
   = odd 2
   = even 1
   = odd 0
   = False
```

Funkcije višeg reda

Funkcije višeg reda

Funkcije koje za argument imaju funkciju ili vraćaju funkciju kao rezultat

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

```
twice :: (a -> a) -> (a -> a)
twice f = f.f
```

```
ghci> twice (*2) 3
12
ghci> twice reverse [1,2,3]
[1,2,3]
```

Zašto su korisne?

Uobičajeni programski idiomi mogu se kodirati kao funkcije u samom jeziku

- Na primer petlje i uslovi

Domenski jezici se mogu definisati kao kolekciju funkcija višeg reda

Alegarske osobine funkcija višeg reda mogu se koristiti za rezonovanje o programima

Funkcija `map`

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

ili

```haskell
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]
ghci> map reverse ["abc","def","ghij"]
["cba","fed","jihg"]
```

Funkcija `filter`

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p []   = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

```
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> filter (/= ' ') "abc def ghi"
"abcdefghi"
```

Funkcija `foldr` (*fold right*)

Mnoge funkcije čiji je argument lista, može se definisati koristeći isti šablon

```
f []       = v
f (x: xs) = x # f xs
```

```
sum [] = 0
sum (x:xs) = x + sum xs

product [] = 1
product (x:xs) = x * product xs
```

```
or [] = False
or (x:xs) = x || or xs

and [] = True
and (x:xs) = x && and xs
```

```
sum :: Num a => [a] -> a
sum = foldr (+) 0

product :: Num a => [a] -> a
product = foldr (*) 1
```

```
or :: [Bool] -> Bool
or = foldr (||) False

and :: [Bool] -> Bool
and = foldr (&&) True
```

Funkcija `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

```
sum = foldr (+) 0
```

```
sum [1,2,3] =
   = foldr (+) 0 [1,2,3]
   = (+) 1 (fold (+) 0 [2,3])
   = (+) 1 ((+) 2 (fold (+) 0 [3]))
   = (+) 1 ((+) 2 ((+) 3 (fold (+) 0 [])))
   = (+) 1 ((+) 2 ((+) 3 (0)))
```

Funkcija `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

```
sum = foldr (+) 0
```

Menjamo svaku pojavu (:) sa (+), a praznu listu sa 0

```
sum [1,2,3] =
   = foldr (+) 0 [1,2,3]
   = foldr (+) 0 (1:(2:(3:[])))
   = 1+(2+(3+0))
   = 6
```

```
product = foldr (*) 1
```

```
product [1,2,3] = ?
```

Funkcija `foldr`

```
length :: [a] -> Int
length = foldr (\_ n -> 1+n) 0
```

```
length [1,2,3] =
    = length (1:(2:(3:[])))
    = 1+(1+(1+0))
    = 3
```

```
reverse :: [a] -> [a]
reverse = foldr (\x -> \xs -> xs ++ [x]) []
```

```
let f = \x -> \xs -> xs ++ [x]
reverse = foldr f []
reverse [1,2,3] =
   = foldr f [] [1,2,3]
   = f 1 (foldr f [] [2,3])
   = f 1 (f 2 (foldr f [] [3]))
   = f 1 (f 2 (f 3 (foldr f [] [])))
   = f 1 (f 2 (f 3 []))
   = f 1 (f 2 [3])
   = f 1 [3,2]
   = [3,2,1]
```

```
reverse [1,2,3] =
    = reverse (1:(2:(3:[])))
    = (([] ++ [3]) ++ [2]) ++ [1]
    = [3,2,1]
```

Funkcija `foldr`

```
foldr (#) v [x0,x1,...,xn] = x0 # (x1 # (... (xn # v) ...))
```

Funkcija `foldl` (*fold left*)

Za levo asocijativne operatore

```
sum :: Num a => [a] -> a
sum = sum' 0
      where
          sum' v [] = v
          sum' v (x:xs) = sum' (v+x) xs
```

```
sum [1,2,3] =
   = sum’ 0 [1,2,3]
   = sum’ (0+1) [2,3]
   = sum’ ((0+1)+2) [3]
   = sum’ (((0+1)+2)+3) []
   = (((0+1)+2)+3)
   = 6
```

```
f v [] = v
f v (x:xs) = f (v # x) xs
```

Funkcija `foldl` (*fold left*)

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Može se predstaviti pomoću jednakosti

```
foldl (#) v [x0,x1,...,xn] = (... ((v # x0) # x1) ...) # xn
```

```
ghci> foldl (/) 64 [4,2,4]
2.0
```

```
foldl (/) 64 [4,2,4] =
  = foldl (/) ((/) 64 4) [2,4]
  = foldl (/) 16 [2,4]
  = foldl (/) ((/) 16 2) [4]
  = foldl (/) 8 [4]
  = foldl (/) ((/) 8 4) []
  = foldl (/) 2 []
  = 2
```

Operator kompozicije

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

`twice f = f . f`	umesto	`twice f x = f (f x)`
`odd n = not (even n)`	umesto	`odd = not . even`

Još neke funkcije višeg reda

Da li svi elementi zadovoljavaju predikat?

```
ghci> all even [2,4,6,8]
True
```

Da li bar neki element zadovoljava predikat?

```
ghci> any odd [2,4,6,8]
False
```

Izdvojiti elemente dok je predikat zadovoljen

```
ghci> takeWhile even [2,4,6,7,8]
[2,4,6]
```

Ukloniti elemente dok je predikat zadovoljen

```
ghci> dropWhile odd [1,3,5,6,7]
[6,7]
```

Napisati definicije prethodnih funkcija

Primer. Glasanje

Kandidat sa najviše glasova pobeđuje

```haskell
import Data.List

votes :: [String]
votes = ["Red", "Blue", "Green", "Blue", "Blue", "Red"]
```

Broj pojavljivanja

```haskell
count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)
```

```
ghci> count "Red" votes
2
```

Primer. Glasanje

Uklanjanje duplikata

```
rmdups :: Eq a => [a] -> [a]
rmdups [] = []
rmdups (x:xs) = x : filter (/= x) (rmdups xs)
```

```
ghci> rmdups votes
["Red","Blue","Green"]
```

Sortirani parovi

```
result :: Ord a => [a] -> [(Int,a)]
result vs = sort [(count v vs, v) | v <- rmdups vs]
```

```
ghci> result votes
[(1,"Green"),(2,"Red"),(3,"Blue")]
```

Pobednik

```
winner :: Ord a => [a] -> a
winner = snd . last . result
```

```
ghci> winner votes
"Blue"
```