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)]

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

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

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

Liste

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

“Peglanje” liste

```
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
```

Liste

```
doubleAll :: [Int] -> [Int]
doubleAll xs = [2*x | x <- xs]
```

```
*Main> doubleAll [2,1,5,8,2,4]
[4,2,10,16,4,8]
```

„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')]
```

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

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

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

```
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]
```

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
```

Beskonačne liste

```
ones :: [Int]
ones =  1 : ones

addFirstTwo :: [Int] -> Int
addFirstTwo (x:y:zs) = x+y

from :: Int -> [Int]
from n = n : from (n+1)

fromStep :: Int -> Int -> [Int]
fromStep n m = n : fromStep (n+m) m
```

```
ones

addFirstTwo ones

from 5

fromStep 5 25
```

Beskonačne liste

```
pytagTriples = [(x,y,z) |
    z <- [2..], y <- [2 .. z-1],
    x <- [2 .. y-1], x*x + y*y == z*z]

primes :: [Int]
primes = sieve [2..]
sieve (x:xs) = x : sieve [y | y <- xs,
                          y `mod` x > 0]
```

```
pytagTriples

primes
```

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 "abbcccddddeeeeee"
[6.666667,13.333334,20.0,26.666668,33.333336,0.0,0.0,0.0
```

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
```

```
ghci> let table' = freqs "kdvnhoo lv ixq"
ghci> [chisqr (rotate n table') table | n <- [0..25]]
[1408.8524,640.0218,612.3969,202.42024,1439.9456,4247.318,650.9992,
```

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

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

Kompozicija

```
fac 4
  = product (take 4 [1..])
  = product [1, 2, 3, 4]
  = 1 * 2 * 3 * 4
  = 24
```

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

Rekurzija

```
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

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

Kompozicija

```
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]
```

Element liste

```
clan :: Eq a => a -> [a] -> Bool
clan x [] = False
clan x (y:ys) = (x == y) || (clan x ys)
```

```
clan 2 [1,2,3,4]
clan 'b' "abc"
```

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`

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

ili

```
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`

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

```
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"
```