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

```
ghci> foldr (\x acc -> even x || acc) False [1,3,5,8,9]
True
```

```
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`

Iteracija

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f
    |  n > 0 = f.iter (n-1) f
    | otherwise = id
```

```
(twice (*2)) 5
(iter 5 (*2)) 1
```

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

```
import Data.List

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

Broj pojavljivanja

```
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"
```

Definisanje tipova i klasa

Deklaracija tipa

Definisanje sinonima za tipove

```
type String = [Char]
```

```
type Pos = (Int,Int)

origin :: Pos
origin = (0,0)

left :: Pos -> Pos
left (x,y) = (x-1, y)
```

```
ghci> left (3,4)
(2,4)
```

Deklaracija tipa

Mogu biti ugnježdene

- Na osnovi jednog tipa može se definisati drugi

```
type Pos = (Int,Int)
type Trans = Pos -> Pos
```

Ne mogu biti rekurzivne

- Ovakvu definiciju nije moguće „razviti“

```
type Tree = (Int, [Tree])
```

Deklaracija tipa

Mogu biti sa parametrom

```
type Pair a = (a,a)
```

```
copy :: a -> Pair a
copy x = (x,x)

mult :: Pair Int -> Int
mult (m,n) = m*n
```

```
ghci> copy 3
(3,3)
```

```
ghci> mult (3,4)
12
```

Mogu imati više parametrara

```
type Assoc k v = [(k,v)]

find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k’,v) <- t, k == k’]
```

```
*Main> find 'c' [('a',5),('b',7),('c',3),('d',12)]
3
```

Deklaracija tipa

```
type Vector =  [Float]
scaleProduct :: Vector -> Vector -> Float


type Matrix = [Vector]
matrixProduct :: Matrix -> Matrix -> Matrix


type Polinom = [(Int, Int)]
vredPolinoma :: Int -> Polinom -> Int
```

Deklaracija podataka

Definicija novog tipa navođenjem vrednosti (konstruktora)

```
data Bool = False | True
```

Mogu se posmatrati kao kontekstno slobodne gramatike

```
data Answer = Yes | No | Unknown
    deriving (Show)

answers :: [Answer]
answers = [Yes, No, Unknown]

flip' :: Answer -> Answer
flip' Yes     = No
flip' No      = Yes
flip' Unknown = Unknown
```

```
ghci> flip' Yes

<interactive>:5:1: error:
    * No instance for (Show Answer) arising from a use of `print'
    * In a stmt of an interactive GHCi command: print it
```

```
ghci> flip' Yes
No
```

Deklaracija podataka

```
data Move = North | South | East | West
   deriving (Show)

move :: Move -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East (x,y) = (x+1,y)
move West (x,y) = (x-1,y)

moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)
```

```
ghci> moves [North,North,West] (1,2)
(0,4)
```

Enumerisani tipovi

```
data Temp = Cold | Hot
    deriving (Eq, Ord, Enum, Show, Read)
data Season = Spring | Summer |
              Autumn | Winter
    deriving (Eq, Ord, Enum, Show, Read)

weather :: Season -> Temp
weather Summer = Hot
weather _ = Cold
```

```
weather Spring
```

Deklaracija podataka

```
data Shape = Circle Float | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y

isRound :: Shape -> Bool
isRound (Circle _) = True
isRound _ = False
```

```
ghci> :type Circle
Circle :: Float -> Shape
ghci> area (square 5)
25.0
```

Vektorski tipovi

```
data People = Person Name Age
type Name = String
type Age = Int

showPerson :: People -> String
showPerson (Person st n) = st ++
                " -- " ++ show n
```

```
showPerson(Person "PeraPeric" 22)
```

Deklaracija podataka

U slučajevima kada se koriste „zaključane“ biblioteke u kojima se definisani neki tipovi podataka

- Jednostavno je definisati nove tipove na osnovu već definisanih
- Jednostavno je definisati nove metode za definisane tipove
- Problem predstavlja definisanje podtipa koji ne može da se doda „zaključanom“ tipu

```
data Shape = ... | Polygon...   !!!
```

Deklaracija podataka sa parametrima

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

```
ghci> safehead []
Nothing
ghci> safehead [2,3,4]
Just 2
```

Unija

```
data Either a b = Left a | Right b
    deriving (Eq, Ord, Show, Read)
```

Pogodno za obradu grešaka

```
safeDivide :: Int -> Int -> Either String Int
safeDivide _ 0 = Left "Cannot divide by zero"
safeDivide x y = Right (x `div` y)
```

```
*Main> safeDivide 10 2
Right 5
*Main> safeDivide 5 0
Left "Cannot divide by zero"
```

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left a)  = f a
either f g (Right b) = g b
```

Rekurzivni tipovi

```
data Nat = Zero | Succ Nat
    deriving (Show)

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Peanovi brojevi

Kako bez Int definisati

```
add :: Nat -> Nat -> Nat
```

```
ghci> int2nat 4
Succ (Succ (Succ (Succ Zero)))
```

Aritmetički izrazi

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
    deriving (Show)

e1 :: Expr
e1 = Add (Val 1) (Mul (Val 3) (Val 3))
```

```
size :: Expr -> Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y
```

```
eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

```
ghci> e1
Add (Val 1) (Mul (Val 3) (Val 3))
ghci> size e1
3
ghci> eval e1
10
```

Kako bi za ovakav tip podataka bila definisana funkcija fold i kako bi se koristila?

Binarna stabla

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

```
occurs :: Eq a => a -> Tree a -> Bool
```

- Pojavljivanje čvora u stablu

```
flatten :: Tree a -> [a]
```

- Formiranje sortirane liste od stabla

Iskazne formule

```
data Prop = Const Bool
          | Val Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

```
isTaut :: Prop -> Bool
```

- Ispitivanje da li je formula tutologija
  - `eval` – vrednost formule
  - `vars` – iskazna slova iz formule
  - `bools` – generisanje svih valuacija
  - `substs` – sve kombinacije vrednosti iskaznih slova i njihovih valuacija

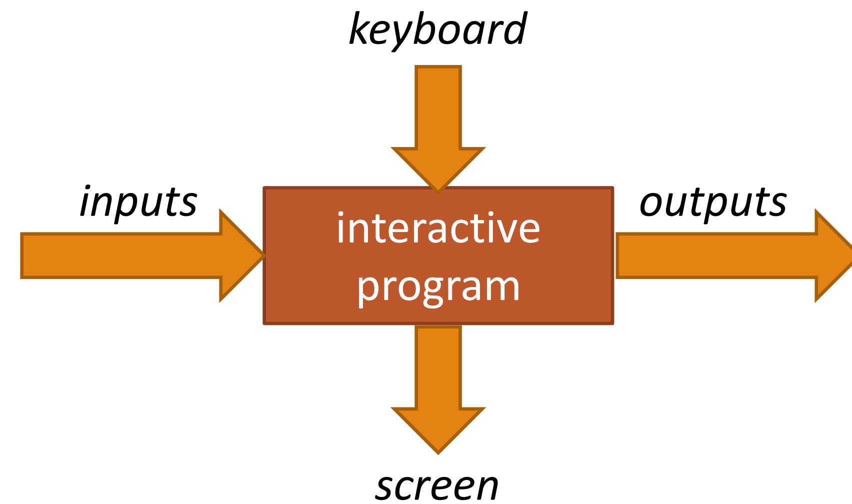
```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False:) bss ++ map (True:) bss
          where bss = bools (n-1)
```

Interaktivni programi

Uvod

Haskell programi su čisto funkcijski/matematički

No side effects



ReadLine – nije matematička funkcija

Interactive programs have side effects

Current “state of world” as argument

Tip IO

```
type IO = World -> World

type IO a = World -> (a, World)
```

IO a – akcija koja vraća tip a

- IO Char – vraća karakter
- IO () -- akcija koja ne vraća vrednost

Standarda biblioteka sadrži veliki broj IO akcija

- getChar :: IO Char <~> getChar :: () -> IO Char
- putChar :: Char -> IO ()
- return :: a -> IO a

Sekvence

Niz akcija koje se spajaju u jednu korišćenjem ključne reči do

```
do v1 <- a1
   v2 <- a2
   ...
   vn <- an
   return (f v1 v2 ... vn)
```

```
a :: IO (Char, Char)
a = do x <- getChar
       getChar
       y <- getChar
       return (x,y)
```

```
ghci> a
a
b
('a','b')
```

Učitavanje i štampanje stringova

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                do xs <- getLine
                   return (x:xs)

b :: IO String
b = do x <- getLine
       return x
```

```
ghci> b
Haskell
"Haskell"
```

Učitavanje i štampanje stringova

```
putStr          :: String -> IO ()
putStr []       = return ()
putStr (x:xs)   = do putChar x
                     putStr xs
```

```
ghci> putStr "abcd"
abcdghci>
```

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

```
ghci> putStrLn "abcd"
abcd
```

Interaktivni program

```
strlen :: IO ()
strlen = do putStr "Enter a string "
            xs <- getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

```
ghci> strlen
Enter a string Haskell
The string has 7 characters
```

Izračunavanje akcija izvršava „*side effects*“ dok je konačni rezultat zanemaren

Hangman

Top-down pristup

```
import System.IO
```

```
hangman :: IO ()
hangman = do putStrLn "Think of a word:"
             word <- sgetLine
             putStrLn "Try to guess it:"
             play word
```

```
*Main> hangman
Think of a word:
----------
Try to guess it:
?beograd
-rag--e-a-
?kragujevac
You got it!
```

Hangman

Prihvatanje reči, ali bez prikazivanja karaktera

```
sgetLine :: IO String
sgetLine = do x <- getCh
              if x == '\n' then
                 do putChar x
                    return []
              else
                 do putChar '-'
                    xs <- sgetLine
                    return (x:xs)
```

Hangman

Prihvatanje karaktera, bez prikazivanja

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           x <- getChar
           hSetEcho stdin True
           return x
```

Hangman

Glavna petlja

```
play :: String -> IO ()
play word = do putStr "?"
               guess <- getLine
               if guess == word then
                 putStrLn "You got it!"
               else
                  do putStrLn (match word guess)
                     play word
```

```
match :: String -> String -> String
match xs ys = [if elem x ys then x else '-'| x <- xs]
```

Kako proširiti brojačem pokušaja?