**Spajanie funkcii**  
f1 (f2 x) = (f1 . f2) x // “f1 po f2”

((f1 . f2) . f3) x = f1(f2(f3 x))

**Lokalna definicia**  
**let** *definicia* **in** *vyraz*   
Pr: let mocnina x = x\*x in mocnina 5 == let mocnina x = x\*x; x=5 in mocnina x

**Zakladne datove typy**  
 -Integer – lubovolne velke cele cisla  
 -Int – cele cisla do velkosti slova procesoru  
 -Float – realne cisla  
 -Fractional – racionalne cisla  
 -Char – znak  
 -String – retazec   
 -Bool – pravdivostna hodnota

**Zlozene typy**  
 -Usporiadane n-tice: (Bool, Int)  
 -Zoznamy: [Int], [Char], [[Char]] /[Char] == String

**Funkcionalne typy**  
 -Integer -> Bool, Float -> Float -> Float  
 Ak su **x1,x2,x3…xn** a **y** nejake typy, tak **x1 -> x2 -> x3 -> … -> xn -> y** je typ vsetkych funkcii s  
 parametrami typu **x1,x2…xn** a funkcnou hodnotou typu **y**.

**Polymorfne typy**  
 Typova premenna: miesto konkretneho typu formalneho parametru sa pouzije typova  
 premenna, pri aplikacii funkcie na konkretne paramatre sa za typovu premennu dosadi  
 typ, ktory zodpoveda pouzitemu parametru.  
 -fst :: (a,b) -> a

**Typove triedy** -Integral – celociselne  
 -Num – numericke  
 -Ord – usporiadatelne  
 -Eq – porovnatelne na rovnost  
 -Priklady typov s obmedzenim typovej premennej: odd :: Integral a => a -> Bool

**Zapis binarnych funkcii**  
Infixovy – 3+4, 4\*5  
Prefixovy (+) 3 4, (\*) 4 5

Arita funkcie = oznacuje pocet parametrov funkcie

**Zoznam**  
 -postupnost hodnot rovnakeho typu v hranatych zatvorkach oddelenych ciarkou  
 -postupnost moze byt prazdna, konecna, nekonecna  
 [3,3,3], [‘a’,’h’,’o’,’j’] == “ahoj”, [[1],[2],[3]]  
 -operator pripojenia prvku na zaciatok zoznamu: ( : ) :: a -> [a] -> [a]  
 3 : [3,3,3] == [3,3,3,3], 1:2:3:[] == [1,2,3]  
 -operator spojenia zoznamov rovnakeho typu: (++) :: [a] -> [a] -> [a]  
 (++) “Ahoj” “svet!” == “Ahoj svet!”

**Funkcie nad zoznamom**  head :: [a] -> a // Prvy prvok zoznamu  
 head (x:\_) = x  
 Pr: head [1,2,3,4,5] == 1

tail :: [a] -> [a] // Zoznam bez prveho prvku zoznamu  
 tail (\_:s) = s  
 Pr: tail [1,2,3,4,5] == [2,3,4,5]

last :: [a] -> a // Posledny prvok zoznamu  
 last (x:[]) = x  
 last (\_:s) = last s  
 Pr: last [1,2,3,4,5] == 5

init :: [a] -> [a] // Zoznam bez posledneho prvku zoznamu  
 init (\_:[]) = []  
 init (x:\_:[]) = [x]  
 init (x:s) = x: init s  
 Pr: init [1,2,3,4,5] == [1,2,3,4]

take :: Int -> [a] -> [a] // Prvych n prvkov zoznamu  
 take n \_ | n <= 0 = []  
 take \_ [] = []  
 take n (x:xs) = x : take (n-1) xs  
 Pr: take 3 [1,2,3,4,5] == [1,2,3]

drop :: Int -> [a] -> [a] // Zoznam bez prvych n prvkov zoznamu  
 drop n xs | n <= 0 = xs  
 drop \_ [] = []  
 drop n (\_:xs) = drop (n-1) xs  
 Pr: drop 3 [1,2,3,4,5] == [4,5]

concat :: [[a]] -> [a] // Spojenie zoznamov v zozname  
 concat [] = []  
 concat (x:s) = x ++ concat s  
 Pr: concat [[1,2],[3,4],[5]] == [1,2,3,4,5]

filter :: (a -> Bool) -> [a] -> [a] // Vynechanie prvkov nesplnajucich danu podmienku  
 filter \_ [] = []  
 filter f (x:s) = if (f x) then x : filter f s  
 else filter f s  
 Pr: filter (>2) [1,2,3,4,5] == [3,4,5]

replicate :: Int -> a -> [a] // Vytvorenie zoznamu k-nasobnym kopirovanim prvku  
 replicate 0 \_ = []  
 replicate k x = x : replicate (k-1) x  
 Pr: replicate 5 1 == [1,1,1,1,1]

takeWhile :: (a -> Bool) -> [a] -> [a] // Vynechanie prvku od prveho, ktory nesplna podmienku  
 takeWhile \_ [] = []  
 takeWhile p (x:s) = if (p x) then x : takeWhile p s  
 else []  
 Pr: takeWhile (<4) [1,2,3,4,5] == [1,2,3]

dropWhile :: (a -> Bool) -> [a] -> [a] // Vynechanie prvkov, po prvy, ktory nesplna podmienku  
 dropWhile \_ [] = []  
 dropWhile p (x:s) = if (p x) then dropWhile p s  
 else x : s  
 Pr: dropWhile (<4) [1,2,3,4,5] == [4,5]

map :: (a -> b) -> [a] -> [b] // Aplikacie funkcie f na vsetky prvky zoznamu  
 map \_ [] = []  
 map f (x:s) = f x : map f s  
 Pr: map (^2) [1,2,3,4,5] == [1,4,9,16,25], Pr2: map (2^) [1,2,3,4,5] == [2,4,8,16,32]

zip :: [a] -> [b] -> [(a,b)] // Spojenie dvoch zoznamov do zoznamu uspor. dvojic  
 zip [] \_ = []  
 zip \_ [] = []  
 zip (x:s) (y:t) = (x,y) : zip s t   
 Pr: zip [1,2,3] [4,5] == [(1,4),(2,5)]

unzip :: [(a,b)] -> ([a],[b]) // Rozdelenie zoznamu dvojich na dvojicu zoznamov  
 unzip [] = ([],[])  
 unzip ((x,y):s) = (x : fst t, y : snd t) where t = unzip s  
 Pr: unzip [(1,2),(3,4),(5,6)] == ([1,3,5],[2,4,6])

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c] // Vypocet aplikacie bin. f na zoznamy argumentov  
 zipWith \_ \_ [] = []  
 zipWith \_ [] \_ = []  
 zipWith f (x:s) (y:t) = f x y : zipWith f s t  
 Pr: zipWith (+) [1,2,3] [4,5,6] == [5,7,9]

Funkcia vyssieho radu: funkcionala, funkcia, ktorej aspon jeden argument alebo jej vysledok je zase funkcia.  
Kazda funkciu, ktora ma aspon 2 argumenty, sa moze chapat ako funkcia vyssieho radu.

(\*) :: Num a => a -> a -> a sa da citat ako (\*) :: Num a => a -> (a -> a)  
Funkcia, ktora berie 2 argumenty typu **a** a vracia hodnotu typu **a** == funkcia, ktora berie hodnotu ciselneho typu **a** a vracia hodnotu typu (a -> a)

Ciastocna aplikacia: ak chapeme n-arne funkcie (n >= 2) ako funkcie vyssieho radu, mozme tieto funkcie ciastocne aplikovat, tj. Vyhodnotit aj pre neuplne zadanie argumentov

Pr: f = (\*) 3  
 f :: Num a => a -> a  
 f 4 == ((\*) 3) 4 == 12

**((^) 3 ) 4 == 3^4 /= (^3) 4 == 4^3**

Ak chceme zabranit ciastocnej aplikacii, musime definovat funkciu tak, aby v jej type bol len jeden vyskyt **->**, pokial chceme funkcii predat viac argumentov, predame ich ako usporiadane n-tice.  
Pr: krat :: Num a => a -> a -> a krat1 :: Num a => (a,a) -> a  
 krat x y = x\*y krat1 (x,y) = x\*y

**Funkcie curry a uncurry**  
 curry :: ((a,b) -> c) -> a -> b -> c  
 curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c  
 uncurry f (x,y) = f x y

Pr: curry krat1 5 4 == krat1 (5,4) == 20 Pr2: uncurry krat (5,4) == krat 5 4 == 20

**Operatorova sekcia**: pre kazdy binarny operator je mozne definovat funkciu, ktora zodpoveda ciastocnej aplikacii funkcie na prvy resp. druhy formalny parameter. Tymto funkciam sa hovori operatorova sekcia.

Pr: (p\*) == (\*) p // Ciastocna aplikacia na prvy argument  
 (\*q) == flip (\*) q // Ciastocna aplikacia na druhy argument

Pr: (1.0/) :: Fractional a => a -> a  
 (`mod` 3) :: Integral a => a -> a  
 (!! 0) :: [a] -> a  
 (>0) :: (Num a, Ord a) => a -> Bool

Bezparametrova definicia funkcie: definicia funkcie bez formalneho parametru  
Pr: f x = (not.odd) x => f = (not.odd)

K vytvoreniu bezparametrovej definicie funkcie pouzivame funkcie: **flip**, **curry**, **uncurry**, **(.)** a **operatorovu sekciu**.

Pr: f x = (3\*x)^7  
 f x = flip (^) 7 (3\*x)  
 f x = flip (^) 7 ((3\*) x)  
 f x = (.) (flip (^) 7) (3\*) x  
 f = (.) (flip (^) 7) (3\*)

**Nepomenovane funkcie**Pr: Globalna definicia a pouzitie funkcie => Lokalna definicia a pouzitie funkcie  
 f x = x\*x =>  
 map f [1,2,3,4,5] == [1,4,9,16,25] => let f x = x\*x in map f [1,2,3,4,5] == [1,4,9,16,25]

Definicia funkcie v mieste jej pouzitia bez uvedenia mena  
Pr: map (\x -> x\*x) [1,2,3,4,5] == [1,4,9,16,25]

**Lambda abstrakcia**: princip vytvarania nepomenovanych funkcii (lambda funkcii).

**Akumulacne funkcie**: “spajaju” jednotlive prvky zoznamu dohromady.

-foldl1 :: (a -> a -> a) -> [a] -> a  
 Pr: foldl1 (-) [1,2,3,4,5] == (((1-2)-3)-4)-5 == -13

-foldr1 (a -> a -> a) -> [a] -> a  
 Pr: foldr1 (-) [1,2,3,4,5] == 1-(2-(3-(4-5))) == 3

**Akumulacne funkcie s inicialnou hodnotou**  
 -foldl :: (b -> a -> b) -> b -> [a] -> b  
 Pr: foldl (-) 6 [1,2,3,4,5] == ((((6-1)-2)-3)-4)-5 == -9

-foldr :: (a -> b -> b) -> b -> [a] -> b  
 Pr: foldr (-) 6 [1,2,3,4,5] == 1-(2-(3-(4-(5-6)))) == -3

Vysledok moze byt znova zoznam  
 Pr: foldr (\x y -> (x+1) : y) [100] [1,2,3,4,5] == 1:(2:(3:(4:(5:[100])))) == [2,3,4,5,6,100]

**Uzivatelom definovane typy**  
 data **Nazov\_typu** = **Datove\_konstruktory** -datove konstruktory sa oddeluju znakom | , nazov typu aj datove konstruktory musia zacinat  
 velkym pismenom  
 Pr: data Dni = Po | Ut | St | Stv | Pia | So | Ne // vsetky datove konstruktory su nularne

Pr: data Farba = RGB Int Int Int // jeden ternarny datovy konstruktor RGB  
 ciastocna aplikacia datoveho konstruktoru:  
 RGB :: Int -> Int -> Int -> Farba  
 RGB 73 :: Int -> Int -> Farba  
 RGB 73 87 :: Int -> Farba  
 RGB 73 87 26 :: Farba

**Typove konstruktory**  
 -nularny typovy konstruktor == typova konstanta  
 -n-arne typove konstruktory, ako napriklad -> alebo [] nedefinuju typ, len predpis ako novy typ  
 vyrobit  
 Parametre (typove premenne) pri n-arnom (n > 0) typovom konstruktore mi umoznia pri  
 definicii n-arnych (n> 0) datovych konstruktoroch pouzit ako parameter typovu premennu,   
 miesto konkretneho typu.

Pr: data Maybe **a** = Nothing | Just a // typova premenna **a** mi umoznuje  
 ze datovy konstruktor **Just** moze ako parameter  
 zobrat hodnotu akehokolvek typu

Pr2: data Maybe2 = Nothing | Just Int // podobny typ, s tymi istymi datovymi  
 konstruktormi, ale bez typovej premennej  
 => datovy konstruktor Just moze ako parameter  
 zobrat hodnotu len typu predom urceneho

**Tvorba typu**  
 -Kazda typova konstanta definuje typ  
 -Typ ziskam tiez uplnou aplikaciou n-arnych typovych konstruktorov na uz definovane typy  
 Pr: (->) Dni Bool = Dni -> Bool  
 Pr2: [] Dni = [Dni]Pr3: (->) (Dni -> Bool) [Dni] = (Dni -> Bool) ->[Dni]

Aplikacia **datovych** konstruktorov vytvara nove hodnoty.  
Aplikacia **typovych** konstruktorov vytvara nove typy.

**Referencna transparentnost**  
 -Vyraz sa vzdy vyhodnoti na rovnaku hodnotu, bez ohladu v akom kontexte je pouzity.  
 -Neda sa napisat program, ktory by spracoval vstup a vyhodnotil sa podla vstupu na   
 rozne hodnoty.  
 -Da sa napisat program, ktory spracuje vstup a podla vstupu vypise rozne hodnoty na vystup.

Vstup-vystupne akcie  
 -Zavedeny specialny typ **IO a** -Vystupne akcie maju typ: **IO ()**  
 -Vstupne akcie maju typ **IO a** // a je typova premenna, ktora nadobuda hodnotu typu podla   
 objektu, ktory vstupuje, Pr: getLine :: IO -> String

getLine :: IO -> a => nacitany retazec sa uchova ako vnutorny vysledok prevedenia tejto  
vstupnej akcie.

**Operator (>>=)**  
 -(>>=) :: IO a -> ( a -> IO b) -> IO b  
 -pomocou tohto operatoru sa pristupuje k vnutornemu vysledku  
 Pr: f >>= g  
 f :: IO a  
 g :: a -> IO b  
 f >>= g :: IO b // Vezme vnutorny vysledok funkcie f a na ten aplikuje funkciu g  
Pr2: getLine >>= putStr // Nacita riadok a ten vypise  
 -Operator (>>=) sa neda pouzit k spojeniu vstupno-vystupnej akcie a funkcie, ktora nieje  
 vstupno-vystupna, pretoze hodnota vyrazu je zavisla na zadanom vstupe  
 => **porusuje to referencnu transparentnost**, je to typovo nespravne.

Pr: getLine >>= length

**Funkcia return** -return :: a -> IO a  
 -prazdna akcia, ktorej prevedenie ma za ciel len naplnit hodnotu vnutorneho vysledku  
 Pr: return [‘A’,’h’,’o’,’j’] >>= putStr

**Operator (>>)**  
 -(>>) :: IO a -> IO b -> IO b  
 -vyraz ma hodnotu poslednej vstup-vystupnej akcie, zabuda hodnotu vnutorneho vysledku  
 Pr: putStr “Jeje” >> putChar ‘!’ // :: IO ()  
 Pr2: getLine >> putStr “nic” // :: IO ()

**Zakladne funkcie pre vystup**  
 putChar :: Char -> IO () // Vypise znak na vystup  
 putStr :: String -> IO () // Vypise retazec na vystup  
 putStrLn :: String -> IO () // Vypise retazec na vystup a odriadkuje  
 print :: Show a => a -> IO () // Vypise hodnotu akehokolvek typu patriaceho do  
 typovej triedy Show na vystup a odriadkuje

**Zakladne funkcie pre vstup**  
 getChar :: IO Char // Nacita znak zo vstupu  
 getLine :: IO String // Nacita riadok zo vstupu  
 getContents :: IO String // Nacita vsetok obsah zo vstupu ako jeden retazec  
 interact :: (String -> String) -> IO ()

**Zakladne funkcie pre pracu so subormi**  
 type FilePath = String // Definuje typovy alias  
 readFile :: FilePath -> IO String // Nacita obsah suboru ako retazec  
 writeFile :: FilePath -> String -> IO () //Zapise retazec do suboru(existujuci obsah vymaze)  
 appendFile :: FilePath -> String -> IO () // Pripise retazec do suboru

const id ‘!’ True :: Bool /= const (id ‘!’) True :: Char

**Typovanie funkcii** (&&) True => (&&) :: Bool -> Bool -> Bool , True :: Bool => (&&) True :: Bool -> Bool  
 const True => const :: a -> b -> a, True :: Bool => const True :: b -> Bool  
 [] : [] : [] => [] :: [a], (:) :: a -> [a] -> [a], [] : [] : [] :: [[a]]  
 ([] : []) : [] => [] :: [a], (:) :: a -> [a] -> [a], ([] : []) : [] :: [[[a]]]

map (filter not)  
 map :: (a->b) -> [a] -> [b]  
 filter :: (c->Bool) -> [c] -> [c]  
 not :: Bool -> Bool   
 (filter not) – (c->Bool) ~ Bool->Bool => c ~ Bool => (filter not) :: [Bool] -> [Bool]  
 map (filter not) – a->b ~ [Bool] -> [Bool] => a~[Bool], b~[Bool] =>   
 map (filter not) :: [[Bool]] -> [[Bool]]

const id ‘!’ True  
 const :: a -> b -> a  
 id :: c -> c  
 ‘!’ :: Char  
 True :: Bool  
 const id – a ~ c->c => const id :: b -> c -> c  
 const id ‘!’ – b ~ Char => const id ‘!’ :: c -> c  
 const id ‘!’ True – c ~ Bool => const id ‘!’ True :: Bool

fst (fst, snd) (snd, fst) (True, False)  
fst :: (a,b) -> a  
snd :: (c,d) -> d  
True :: Bool  
False :: Bool  
fst (fst,snd) – a ~ (a,b) -> a, b ~ (c,d) -> d => fst (fst,snd) :: (a,b) -> a  
fst (fst,snd) (snd,fst) – a ~ (c,d) -> d, b ~ (a,b) -> a => fst (fst,snd)(snd,fst) :: (c,d) -> d  
fst (fst, snd) (snd,fst)(True,False) – c ~ Bool, d ~ Bool => fst(fst,snd)(snd,fst)(True,False) :: Bool

head [head] [tail] [[]]  
head :: [a] -> a  
[head] :: [[b] -> b]  
[tail] :: [[c] -> [c]]  
[[]] :: [[d]]  
head [head] == head [[b]->b] -> [a] -> a => head [head] :: [a] -> a  
head [head] [tail] == [[c] -> [c]] -> [c] -> [c] => head [head] [tail] :: [c] -> [c]  
head [head] [tail] [[]] :: [[d]]

f1 :: (a -> b) -> [a] -> [b], f2 :: (c -> d -> e)

f1 f2 :: [c] -> [d -> e]  
Vysvetlenie: (a -> b) ~ (c -> d -> e) => a ~ c, b ~ (d -> e)

const (id(id)) :: b -> a -> a /= const id(id) :: a -> a

Pr: map filter   
 map :: (a -> b) -> [a] -> [b]  
 filter :: (c -> Bool) -> [c] -> [c]  
 map filter => (a -> b) ~ (c -> Bool) -> [c] -> [c] => a ~ (c -> Bool), b ~ [c] -> [c]  
 map filter :: [c -> Bool] -> [[c] -> [c]]

Pr: const map  
 const :: a -> b -> a  
 map :: (c -> d) -> [c] -> [d]  
 const map => a ~ (c -> d) -> [c] -> [d]  
 const map :: b -> (c -> d) -> [c] -> [d]

Pr: map const  
 map :: (a -> b) -> [a] -> [b]  
 const :: c -> d -> c  
 map const => (a -> b) ~ c -> d -> c => a ~ c, b ~ d -> c  
 map const :: [c] -> [d -> c]

**Zapis zoznamu**

1.Vypisom

a) Prosty vypis: 1:2:3:4:5:[] == [1,2,3,4,5]   
b) Hromadny vypis:   
 enumFrom m = [m…] Pr: enumFrom 0 == [0,1,2,3,4,5... ∞]  
 enumFromTo m n == [m…n] Pr: enumFromTo 0 5 == [0,1,2,3,4,5]  
 enumFromThen m m´ == [m,m´…] Pr: enumFromThen 0 4 == [0,4,8,12,16… ∞]  
 enumFromThenTo m m´ n ==[m,m´...n] Pr: enumFromThenTo 0 4 20 == [0,4,8,12,16,20]

2.Intenzionalnym zapisom  
 [definicny\_vyraz | generator a kvalifikatory]

generator == nova\_premenna <- zoznam alebo vzor <- zoznam  
 - definuje novu premennu pouzitelnu bud v definicnom\_vyraze alebo v ktoromkolvek  
 kvalifikatore napravo od jeho definicie  
 - nova premenna postupne nadobuda hodnoty z daneho zoznamu  
 - pri pouziti viacerych generatorov sa robia vsetky ich kombinacie, poradie kombinacii je   
 dane prioritou generatorov v definicnom vyraze (viac vlavo => vacsia priorita)

predikat== vyraz typu Bool  
 - moze vyuzivat premenne definovane nalavo od predikatu  
 - vygenerovane instancie, nevyhovujuce predikatu nebudu brane v uvahu pri definicii  
 vysledneho zoznamu

Pr: [n^2 | n <- [0..3]] == [0,1,4,9]   
Pr2: [(m,n) | m <- [1..3], n <- [1..3], n <= m] == [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]  
Pr3: [replicate n m | n <- [2,3], m <- [“xyz”]] == [“xx”, “yy”, “zz”, “xxx”, “yyy”, “zzz”]  
Pr4: Redefinicia length => length2 s = sum [1 | \_ <- s]

QuickSort – zoradenie usporiadatelnych hodnot konecneho zoznamu vzostupne (<)  
qSort :: Ord a => [a] -> a  
qSort [] = []  
qSort (p:s) = qSort [x | x <- s, x<p] ++[p]++ qSort [x | x <- s, x>=p]

QuickCheck :: prop => prop -> IO ()  
 - import Test.QuickCheck  
 - generuje nahodne hodnoty a na tych testuje zadanu funkciu  
 - quickCheck (\x -> zadana\_funkcia x == ocakavany vysledok)  
Pr: prop1 :: Int -> Bool  
 prop1 x = (x+1)\*(x+1)==x\*x + 2\*x + 1

=>quickCheck prop1 alebo test s vypisovanim jednotlivych hodnot == verboseCheck prop1

foldr1 :: (a -> a -> a) -> [a] -> a  
foldr :: (a -> b -> b) ->b -> [a] -> b  
foldl :: (b -> a -> b) -> b -> [a] -> b  
(.) :: (b -> c) -> (a -> b) -> a -> c  
curry :: ((a,b) -> c) -> a -> b -> c  
uncurry :: (a -> b -> c) -> (a,b) -> c  
flip :: (a -> b -> c) -> b -> a -> c  
(>>=) :: IO a -> ( a -> IO b) -> IO b  
(>>) :: IO a -> IO b -> IO b  
(+) = inflixl 6, (\*) = infixl 7, (^) = infixr 8, (.) = infixr 9, (>,<,<=,>=,==) = infix 4, (>>=,>>) = infixl 1, (div,mod) = infixl 7