

PENOPORA: Haskell výpisky



1 Základy

1.1 Vlastnosti jazyka

- čistě funkcionální, silně staticky typovaný
- lazy evaluation (líné vyhodnocování výrazů)
- case sensitive

1.2 Interpret a překladač

- GHC nebo Hugs
- název zdrojového souboru - *mojefunkce.hs*
- načtení souboru - *:l mojefunkce*
- znovunačtení souboru - *:r*
- ověření datového typu - *:t*
- nastavení promptu - *:set prompt "ghci> "*

1.3 Komentáře

```
-- radkový komentář  
{- blokový komentář -}
```

1.4 Základní aritmetika

```
2+5          -- scitání  
49*100       -- násobení  
255-124      -- odčítání  
5/2          -- dělení  
10^2         -- druhá mocnina  
3*(-5)       -- záporné číslo v závorkách  
50 * (100 - 4999) -- složený výraz  
True && False  -- logické and  
True || False  -- logické or  
not True      -- negace  
5==5         -- rovnost  
5/=5         -- nerovnost
```

1.5 Volání funkcí

Závorky určují prioritu. Priorita aplikace je nejvyšší.

```
succ 6        -- vrátí následníka, argumenty odděleny mezerou  
pred 6        -- vrátí předchůdce  
min 100 10    -- vrátí menší ze dvou prvků  
max "ahoj" "cau" -- vrátí větší ze dvou prvků  
succ 9 * 10    -- vrátí následníka čísla 9 a poté se násobí 9, tedy 100  
succ (9 * 10)  -- vrátí následníka součinu 9*10, tedy 91  
div 92 10     -- prefixový zápis  
92 `div` 10   -- infixový zápis prefixového  
(+) 2 5       -- prefixový zápis infixového  
odd 1         -- vrátí True pokud je číslo liché  
even 2        -- vrátí True pokud je číslo sudé  
error "Chyba!" -- vyvolání výjimky
```

1.6 Definice funkcí

Názvy proměnných a funkcí začínají malým písmem. V názvu funkcí se může vyskytovat znak apostrofu '. Argumenty se oddělují mezerami. Na pořadí definic funkcí nezáleží, ale pořadí je použito při vyhledávání vzoru pro unifikaci. V interaktivním režimu GHCi je potřeba použít pro definici klíčové slovo *let*.

```
doubleMe x = x + x           -- definice vlastní funkce v souboru
doubleUs x y = doubleMe x + doubleMe y -- definice vlastní funkce v souboru
conanO'Brien = "Ahoj, ja jsem Brian!" -- definice (pojmenovani) v souboru
```

1.7 Podmínky

Podmínky se mohou vyskytovat téměř kdekoli. *Else* je povinný.

```
doubleSmallNumber x = if x > 100    -- prikaz if, else je povinne
                        then x
                        else x*2

list = [if 5 > 3 then "Bla" else "Ble", if 'a' > 'b' then "Neco" else "Nic"]
numb = 4 * (if 10 > 5 then 10 else 0) + 2
```

1.8 Seznamy a řetězce

Seznam je homogenní struktura. Řetězce jsou také seznamy. Seznamy lze do sebe vnořovat. Seznamy lze porovnávat pomocí operátorů <, <=, >, >=, ==, /=. Haskell podporuje nekonečné seznamy (díky vlastnosti lazy evaluation).

Konstruktor je `:`.

```
let lostNumbers = [4,8,15,16,23,42]      -- definice seznamu
[1,2,3] ++ [4,5,6]                       -- spojení seznamu
"nazdar " ++ "chlape"                   -- spojení řetězce
1:[2,3]                                 -- přidání prvku na začátek seznamu
1:2:3:[]                                -- přidání více prvků do seznamu
'a':"hoj"                                -- přidání znaku na začátek řetězce
[4,8,15,16,23,42] !! 0                   -- získání prvního prvku seznamu
"nazdar" !! 2                           -- získání třetího znaku řetězce

let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]] -- definice vnoreného seznamu
b ++ [[1,1,1,1]]                          -- spojení vnorených seznamu
[6,6,6]:b                                  -- přidání seznamu na začátek
b !! 2                                     -- získání třetího prvku

head [4,8,15,16,23,42]                    -- vrátí první prvek seznamu
tail [4,8,15,16,23,42]                    -- vrátí vše mimo první prvek seznamu
last [4,8,15,16,23,42]                    -- vrátí poslední prvek seznamu
init [4,8,15,16,23,42]                    -- vrátí vše mimo poslední prvek seznamu
length [4,8,15,16,23,42]                  -- vrátí délku seznamu
null [4,8,15,16,23,42]                    -- vrátí True pokud je seznam prázdný
reverse [4,8,15,16,23,42]                 -- obrátí seznam
take 3 [4,8,15,16,23,42]                  -- vybere daný počet prvků ze začátku seznamu
drop 3 [4,8,15,16,23,42]                  -- zahodí daný počet prvků a vrátí zbytek seznamu
minimum [4,8,15,16,23,42]                 -- vrátí nejmenší prvek
maximum [4,8,15,16,23,42]                 -- vrátí největší prvek
sum [4,8,15,16,23,42]                     -- vrátí součet všech prvků seznamu
product [4,8,15,16,23,42]                 -- vrátí součin všech prvků seznamu
elem 8 [4,8,15,16,23,42]                  -- vrátí True pokud je daný prvek v seznamu

['a'..'z']                                -- rozsah znaků
[1..20]                                    -- rozsah čísel
[2,4..20]                                  -- rozsah čísel s přírůstkem
[20,19..1]                                 -- klesající rozsah čísel
[1..]                                       -- nekonečný seznam
take 10 [0,2..]                             -- prvních 10 násobků čísla 2
cycle [1,2,3]                               -- cyklení seznamu do nekonečna
repeat 5                                    -- cyklení prvku do nekonečna
take 12 (cycle [1,2,3])                     -- prvních 12 prvků nekonečného seznamu
replicate 3 10                             -- vrátí tři výskyty čísla 10 v seznamu
```

1.8.1 Generátory seznamů

```
-- intenzionalni zapisy mnozin (filtrovani)
[x*2 | x <- [1..5]] -- vrati 5 sudych cisel
[x*2 | x <- [1..10], x*2 >= 12] -- [12,14,16,18,20]
[x | x <- [50..100], x `mod` 7 == 3] -- [52,59,66,73,80,87,94]
[x | x <- [10..20], x /= 13, x /= 15, x /= 19] -- [10,11,12,14,16,17,18,20]
[if x<10 then "A" else "B" | x <- [7..13], odd x] -- ["A","A","B","B"]
[x*y | x <- [10,100], y <- [1,2,3]] -- [10,20,30,100,200,300]
[0 | x <- [0..9]] -- [0,0,0,0,0,0,0,0,0,0]
[(m,n) | m <- [1..3], n <- [0,1]] -- [(1,0),(1,1),(2,0),(2,1),(3,0),(3,1)]

-- kombinovani slov
let birds = ["jestrab", "holub", "sokol"]
let adjectives = ["bystry", "chytry", "krasny"]
[adjective ++ " " ++ bird | adjective<-adjectives, bird<-birds]

-- vlastni funkce length, _ znaci ze prvek ze seznamu nepotrebujeme
let length' xs = sum [1 | _ <- xs]

-- odstrani z retezce znaky, které nejsou povoleny v nazvu identifikatoru
let identifStr xs = [x | x <- xs, elem x ['A'..'z'] || elem x ['0'..'9'] || x == '_']

-- fitrovani vnorených seznamu
let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
[ [ x | x <- xs, even x ] | xs <- xxs]
```

1.9 N-tice

N-tice je heterogenní struktura. Tvoří ji pevně daný a neměnný počet prvků. Konstruktor je `,`.

```
let person = ("Chuck", "Norris", 70) -- definice n-tice
fst (8,11) -- vrati prvni slozku dvojice
snd (True,False) -- vrati druhou slozku dvojice
zip [1,2,3] ["one","two","three"] -- vrati seznam dvojic
zipWith (+) [1,2,3] [3,2,1] -- provede operaci nad seznamy a vrati seznam
zipWith (**) (replicate 10 5) [1..10]

-- seznam pravouhlych trojuhelniku s obvodem 24 jejichz strany jsou mensi nez 10
let triangles = [(a,b,c) | c<-[1..10], b<-[1..c], a<-[1..b], a^2+b^2==c^2, a+b+c==24]
```

2 Datové typy

Haskell má statický typový systém. Umí odvozovat typy. Přehled základních typů:

- Int – celá čísla ohraničená (obvykle maximum 2147483647 a minimum -2147483648)
- Integer – celá čísla neohraničená
- Float – reálná čísla
- Double – reálná čísla s větší přesností
- Char – znak
- Bool – logický typ (hodnoty True, False)
- Ordering – porovnávání (hodnoty GT, LT, EQ)

2.1 Typy výrazu v GHCi

```
:t 'a'           -- 'a' :: Char
:t True          -- True :: Bool
:t "Nazdar"      -- "Nazdar" :: [Char]
:t (True, 'a')   -- (True, 'a') :: (Bool, Char)
```

2.2 Typy funkcí

Explicitní deklarace typu funkce. Jména typů, typových tříd a datové konstruktory se zapisují velkým počátečním písmenem. Ostatní literály se zapisují malým počátečním písmenem.

```
removeNonUppercase :: [Char] -> [Char] -- [Char] je synonymum ke String
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]

addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

2.3 Typové proměnné

```
:t head          -- head :: [a] -> a
:t fst            -- fst :: (a, b) -> a
:t (:)           -- (:) :: a -> [a] -> [a]
:t ((:))3        -- ((:))3 :: (Num t) => [t] -> [t]
:t []            -- [] :: [a]
:t (,,)          -- (,,) :: a -> b -> c -> (a, b, c)
```

2.4 Typové třídy

Třídy definují určité chování. Údaje před symbolem `=>` se nazývají typová omezení. Např. ve funkci `(==)` typ dvou hodnot musí být instancí třídy *Eq*. Přehled základních typových tříd:

- *Eq* – typy podporující testování rovnosti (funkce implementované v této třídě: `==`, `/=`)
- *Ord* – typy podporující porovnávání (funkce: `<`, `<=`, `>`, `>=`, `max`, `min`, `compare`)
- *Show* – převod na řetězec (funkce: `show`)
- *Read* – převod řetězce na typ (funkce: `read`)
- *Enum* – sekvenčně seřazené typy (funkce: `succ`, `pred`)
- *Bounded* – horní a spodní ohraničení (funkce: `minBound`, `maxBound`)
- *Num* – numerická typová třída, obsahuje celá a reálná čísla
- *Integral* – numerická typová třída, obsahuje pouze celá čísla
- *Floating* – numerická typová třída, obsahuje pouze čísla s plovoucí desetinnou čárkou

```
:t (==)          -- (==) :: (Eq a) => a -> a -> Bool
:t elem          -- elem :: (Eq a) => a -> [a] -> Bool
:t read          -- read :: (Read a) => String -> a
:t 20            -- 20 :: (Num t) => t

5==5             -- True
"Nazdar" /= "Cau" -- True

5 >= 2           -- True
5 `compare` 3    -- GT

show 5.334       -- "5.334"

read "True" || False -- True
read "5" - 2       -- 3

-- použití explicitní typové anotace pomoci ::
read "3" :: Int    -- 3
read "3" :: Float  -- 3.0

succ 'B'          -- 'C'

minBound :: Int    -- -2147483648
maxBound :: Char   -- '\1114111'
maxBound :: (Bool, Int, Char) -- (True, 2147483647, '\1114111')

20 :: Int          -- 20
20 :: Float        -- 20.0
```

2.5 Definice typových tříd

```
-- definice typove tridy Eq
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x == y = not (x/=y)
    x /= y = not (x==y)

-- instance typove tridy Eq pro cela cisla
instance Eq Int where
    (==) = primEqInt -- vazba na vestavenou funkci

-- explicitni definice operace rovnosti pro n-tice
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (xx,yy) = if x==xx then y==yy else False

-- definice rovnosti dvou seznamu
instance Eq a => Eq [a] where
    [] == [] = True
    (x:xs) == (y:ys) = x==y && xs==ys
    _ == _ = False

-- instance typove tridy Eq pro TrafficLight
data TrafficLight = Red | Yellow | Green
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False

instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

2.6 Odvozené typové třídy

```
-- trida zahrnujici operatory pro porovnani na usporadani
class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a

    compare x y | x==y = EQ | x<=y = LT | otherwise = GT

    x <= y = compare x y /= GT
    x < y = compare x y == LT
    x >= y = compare x y /= LT
    x > y = compare x y == GT

    max x y | x <= y = y
             | otherwise = x

    min x y | x <= y = x
            | otherwise = y

-- vytvoreni monolitickeho pole: array :: (lx a) => (a,a) -> [(a,b)] -> Array a b
class (Ord a) => lx a where
    -- squares = array (1,10) [(i,i*i)|i<-[1..10]]
    range :: (a,a) -> [a] -- range (0,4) ~> [0,1,2,3,4]
    index :: (a,a) -> a -> Int -- index (1,9) 2 ~> 1
    inRange :: (a,a) -> a -> Bool

-- vicenasobna dedicnost
class (Real a, Fractional a) => RealFrac a where
    ...
```


2.7 Typová synonyma

Příkaz **type** slouží k přejmenování existujícího datového typu.

```
-- typova synonyma
type String = [Char]
type ComplexF = (Float, Float)
type PhoneBook = [(String,String)]
type Matrix a = [[a]]
type AssocList k v = [(k,v)]

-- vyuziti ve funkcich
n = (2,5) :: ComplexF

sumz :: ComplexF -> Float
sumz xs = fst xs + snd xs

conc :: Matrix a -> [a]
conc [] = []
conc (xs:xss) = xs ++ (conc xss)
```

2.8 Jednoduché uživatelské datové typy

Příkaz **newtype** slouží k zabalení existujícího datového typu. Na rozdíl od typových synonym není nutné v programu explicitně přidávat typovou signaturu. Derivace (odvozené instance) pomocí klíčového slova **deriving** lze užít u typových tříd Eq, Ord, Enum, Read, Show a Bounded.

```
-- uzivatelske typy
newtype ComplexC = ReIm (Float,Float)
newtype MatrixC a = Matrix [[a]]

-- instance Show pro zobrazeni typu, napr. ReIm (1.1,2.2)
instance Show ComplexC where
    showsPrec p (ReIm (f1,f2)) = (\r -> "ReIm " ++ sf1 ++ " " ++ sf2 ++ r)
                                   where sf1 = show f1
                                           sf2 = show f2

-- efektivnejsi alternativa
instance Show ComplexC where
    showsPrec p (ReIm (f1,f2)) = ("ReIm " ++) . sf f1 . (' ':'') . sf f2
                                   where sf = showsPrec 9

-- uzivatelsky typ s derivaci typovych trid
newtype ComplexC = ReIm (Float,Float)
    deriving (Show, Read, Eq)
```

2.9 Komplexní datové typy

Příkaz **data** slouží k vytvoření nového datového typu. Parametry typu a klauzule **deriving** jsou nepovinné. Konstruktory jsou funkce, které mají určitou hodnotu.

```
-- obecne schema
data Nazev_typu a1 a2 ... an
    = Konstruktor_1
    | Konstruktor_2
    ...
    | Konstruktor_m
    deriving (...)
```

2.9.1 Výčtové typy

```
-- typ Boolean
data Bool = False | True

-- typ Color a hodnoty Red, Green, Blue
data Color = Red | Green | Blue

isRed :: Color -> Bool
isRed Red = True
isRed _ = False
```

2.9.2 Rozšířené typy

```
-- typ Color s rozšířeným datovým konstruktorem Grayscale
data Color' = Red' | Green' | Blue' | Grayscale Int

getLevelOfGray (Grayscale n) = n
getLevelOfGray _ = 0

-- typ teplota
data Teplota = Nula | Celsius Float | Kelvin Float

zobraz :: Teplota -> String
zobraz Nula = "0"
zobraz (Celsius x) = show x
zobraz (Kelvin x) = show (x + 273.15)

toKelvin :: Teplota -> Float
toKelvin Nula = 273.15
toKelvin (Kelvin x) = x
toKelvin (Celsius x) = x + 273.15

mrzne :: Teplota -> Bool
mrzne Nula = True
mrzne (Kelvin x) = x <= 273.15
mrzne (Celsius x)
  | x <= 0 = True
  | otherwise = False

-- typ geometrického tvaru, kruh má souřadnice a radius, obdélník má souřadnice 2 bodů
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
  deriving (Show)

Circle :: Float -> Float -> Float -> Shape
Rectangle :: Float -> Float -> Float -> Float -> Shape

surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)

map (Circle 10 20) [4,5,6,6] -- konstruktory se chovají jako funkce

-- vylepšený typ geometrického tvaru s použitím typu point
data Point = Point Float Float deriving (Show) -- konstruktor se může jmenovat jako typ
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)

surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)

nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a) (y1+b))
  (Point (x2+a) (y2+b))
```

```

-- typ Den, odvozuje vsechny mozne typove tridy
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
          deriving (Eq, Ord, Show, Read, Bounded, Enum)

show Wednesday          -- prevod typu na retezec (Show)
read "Saturday" :: Day   -- prevod retezce na typ (Read)

Saturday == Sunday       -- False, rovnost (Eq)
Saturday == Saturday     -- True, rovnost (Eq)
Saturday > Friday        -- True, porovnani (Ord)
Monday `compare` Wednesday -- LT, porovnani (Ord)

minBound :: Day          -- Monday, minimum (Bounded)
maxBound :: Day          -- Sunday, maximum (Bounded)

succ Monday              -- Tuesday, naslednik (Enum)
pred Saturday            -- Friday, predchudce (Enum)
[Thursday .. Sunday]     -- [Thursday, Friday, Saturday, Sunday], rada (Enum)
[minBound .. maxBound] :: [Day] -- rada vseh (Enum)

```

2.9.3 Záznamy

Záznamy automaticky vytvoří funkce, pomocí nichž lze přistupovat k prvkům záznamu. Při tvorbě záznamu není potřeba dodržovat pořadí prvků jako u běžných typů. Záznamy jsou vhodné v případech, kdy není zcela jasné pořadí prvků v konstruktoru.

```

-- typ Student
data Student = Student { jmeno :: String
                        , stip :: Int
                        , phd :: Bool
                        } deriving (Eq, Show, Read) -- odvozuje tridu pro porovnani...

let s = Student { jmeno="peno", stip=6700, phd=False } -- vytvoreni studenta
s == Student { jmeno="dusan", stip=6700, phd=True }   -- porovnani studentu
"Student info: " ++ show s                            -- prevod typu na retezec
read "Student {jmeno=\"peno\",stip=6700,phd=False}" :: Student -- prevod retezce na typ

phdPayRise = map (\student -> if phd student
                              then student(stip=(stip student)*2)
                              else student)

```

2.9.4 Parametrické typy

V deklaraci datového typu se nikdy nepoužívají typové třídy.

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

-- typ Color
data RGBColor a = RGBc a a a
data CMYColor a = CMYc a a a
data Color a
  = RGB (RGBColor a)
  | CMY (CMYColor a)
  | Grayscale a

rgb2grayscale :: (Fractional t) => Color t -> Color t
rgb2grayscale (RGB (RGBc r g b)) = Grayscale ((2*r+4*g+2*b)/8)

-- typ 3D Vector
data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n

-- typ vice-dimenzionalni Vector
data Vector a = Vec Int [a]
  deriving (Read, Show, Eq)

initVec :: [a] -> Vector a
initVec l = Vec (length l) l
-- vytvoreni vektoru ze seznamu
-- Vec 4 [1,2,3,4]

dotProd :: (Num a) => Vector a -> Vector a -> a
dotProd (Vec len1 vec1) (Vec len2 vec2) =
  if len1 /= len2
  then error "Bad size!"
  else foldl1 (+) (zipWith (*) vec1 vec2)
-- skalarni soucin

-- typ Matrix
data Matrix a = Mat Int Int [Vector a]
  deriving (Read, Show, Eq)

initMat ll = Mat (length ll) x (map initVec ll)
  where x = chl (length(head ll)) (tail ll)
        chl l [] = 1
        chl l (xs:xss) = if l == length xs
                          then chl l xss
                          else error "Bad Columns"

mulMat (Mat r1 c1 l1) (Mat r2 c2 l2) =
  if c1 /= r2
  then error "Bad size!"
  else initMat form
    where ...
          ...
          form = ...
```

2.9.5 Rekurzivní typy a vlastní operátory

U datových konstruktorů vlastních operátorů musí být vždy obsažena dvojtečka `:`. Priorita je od 0 – 9. Asociativita může být: infixl (+), infixr (:), infix (==). Pattern matching porovnává konstruktory typů.

```
data Stack a = Top a (Stack a) | Bottom

-- vlastní operator s definovanou asociativitou a prioritou
infixr 5 :>
data Stack a = a :> (Stack a) | Bottom deriving (Eq, Show)

-- použití ve funkci
push :: a -> Stack a -> Stack a
push n sx = n :> sx -- push 4 (3 :> 2 :> 1:> Bottom)

-- konstruktor Bottom nahrazen za operator
data Stack' a
  = a :>> (Stack' a)
  | (:||)
  deriving (Eq, Show)
```

```
-- typova promenna a umoznuje tvorit seznamy nad libovolnym DT
-- díky rekurzi může seznam obsahovat neomezené prvku, Cons je konstruktor seznamu (:)
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)

-- příklady vytvoření seznamu
Empty
3 `Cons` (4 `Cons` (5 `Cons` Empty))

-- vlastní operator konstruktoru seznamu
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)

-- vlastní operator konkatenace seznamu
infixr 5 .++
(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)

-- příklad vytvoření seznamu a konkatenace
let a = 3 :-: 4 :-: 5 :-: Empty
a .++ a
```

```
-- výrazy
data Expr = Lit Int | Var Var | Op Ops Expr Expr
data Ops = Add | Sub | Mul | Div | Mod
type Var = Char

-- paměť
newtype Store = Sto (Var -> Int)

initial :: Store
initial = Sto (\v -> 0)

value :: Store -> Var -> Int
value (Sto sto) v = sto v

update :: Store -> Var -> Int -> Store
update (Sto sto) v n = Sto (\w -> if v == w then n else sto w)
```

```

-- binarni vyhledavaci strom
data Tree a = Leaf | Node a (Tree a) (Tree a)
    deriving (Show, Read, Eq)

inorder :: Tree a -> [a]
inorder Leaf = []
inorder (Node d l r) = inorder l ++ d : inorder r

preorder Leaf = []
preorder (Node d l r) = d:preorder l ++ preorder r

postorder Leaf = [ ]
postorder (Node d l r) = postorder l ++ postorder r ++ [d]

height :: (Num a, Ord a) => Tree a -> a
height Leaf = 0
height (Node _ l r) = 1 + max (height l) (height r)

showTree :: Show a => Tree a -> String
showTree Leaf = "*"
showTree (Node d l r) = show d ++ "<" ++ showTree l ++ "," ++ showTree r ++ ">"

-- priklady volani funkci:
-- inorder Leaf
-- inorder (Node 1 Leaf Leaf)
-- inorder (Node 1 (Node 0 (Node (-1) Leaf Leaf) Leaf) (Node 2 Leaf Leaf))

treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right

singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x Leaf = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)

treeToList :: Ord a => Tree a -> [a]
treeToList Leaf = []
treeToList (Node x l r) = (treeToList l) ++ x:(treeToList r)

leafCount :: Tree a -> Int
leafCount Leaf = 1
leafCount (Node _ x y) = (leafCount x) + (leafCount y)

-- vlastni zobrazovaci funkce, rika jak se bude typ zobrazovat, misto deriving Show
data Tree a = Leaf | Node a (Tree a) (Tree a)
instance Show a => Show (Tree a) where
    show t = showTree t

```

```

-- relacni database
data Attribute = St String | Num Int | Bool Bool | Null      -- typy dat v tabulce
type Tuple = [Attribute]                                     -- radek tabulky
type Schema = [String]                                       -- jmena sloupce
type Table = (Schema, [Tuple])                               -- tabulka tvorena schematem a radky
type Database = [(String, Table)]                           -- database je seznam pojmenovanych tabulek

```

```

-- definice index-sekvencniho vyhledavaciho stromu
data ISTree a b = Struct a a Int (STree a b)
    deriving (Show,Eq)

data STree a b = Data a a [(a,b)]
    | Index a a [STree a b]
    deriving (Show,Eq)

-- inicializace stromu: parametr dolni a horni mez indexu,
-- hloubka indexove struktury a pocet polozek v urovni
-- priklad: initIST (1, 100) 2 3
initIST :: Integral a => (a,a) -> Int -> a -> ISTree a b
initIST (l,h) depth items =
    Struct l h depth (mkST l h depth)
    where
        mkST l h 0 = Data l h []
        mkST l h n = Index l h (splitI mkST items l h n 0)

--rozdeli prideleny interval indexu i vytvori zadany pocet polozek v dane urovni
splitI :: Integral a =>
    (a -> a -> Int -> STree a b) -> a -> a -> a -> Int -> a -> [STree a b]
splitI makeST items low high n x =
    if x==items then []
    else makeST newl newh (n-1) : splitI makeST items low high n (x+1)
    where int  = (high-low+1) `div` items
          newl = low + int * x
          newh = if x == items-1 then high else newl + int - 1

```

```

-- binarni vyhledavaci strom s klicem v tride Ord
data Tree key dat = Leaf | Node key dat (Tree key dat) (Tree key dat)
    deriving (Show,Eq)

-- vyvazeni stromu
sameLevel Lf = Lf
sameLevel tree = procList $ {- fsq $ -} inorder tree
    where inorder Lf = []
          inorder (Nd k d l r) = (inorder l) ++ ((k,d) : inorder r)
          mid list = splitAt (length list `div` 2) list
          procList [] = Lf
          procList [(k,d)] = Nd k d Lf Lf
          procList list = Nd k d (procList l) (procList r)
              where
                  (l,((k,d):r)) = mid list

```

```

-- reprezentace Lambda kalkulu
type Variable = Char
data LambdaExpr = Var Variable -- x
                | Appl LambdaExpr LambdaExpr -- (X Y)
                | Abstr Variable LambdaExpr -- (\x.Y)
                deriving (Show, Eq)

-- odstrani duplikaty ze zadaneho seznamu
remDupl :: Eq a => [a] -> [a]
remDupl [] = []
remDupl (x:xs) = if elem x xs then remDupl xs else x:(remDupl xs)

-- zjistí všechny volné proměnné v zadaném lambda výrazu
unboundVars :: LambdaExpr -> [Variable]
unboundVars x = remDupl $ impl x []
    where impl (Var v) xs = if not $ elem v xs then [v] else []
          impl (Appl e1 e2) xs = (impl e1 xs) ++ (impl e2 xs)
          impl (Abstr v e) xs = impl e (v:xs)

-- zjistí všechny vázané proměnné v zadaném lambda výrazu
boundVars :: LambdaExpr -> [Variable]
boundVars x = remDupl $ impl x []
    where impl (Var v) xs = if elem v xs then [v] else []
          impl (Appl e1 e2) xs = (impl e1 xs) ++ (impl e2 xs)
          impl (Abstr v e) xs = v:(impl e (v:xs))

-- overení platnosti substituce
isValid e e' v = eval e []
    where fv' = freeVars e'
          eval (Var w) bs = myelem w bs || not (v==w && intersect fv' bs)
          eval (App e1 e2) bs = eval e1 bs && eval e2 bs
          eval (Abstr w ew) bs = eval ew (w:bs)
myelem x l = filter (==x) l /= []
intersect s1 s2 = foldr (||) False (map (\e -> myelem e s2) s1)

-- provede danou alfa redukci
alphaRed :: LambdaExpr -> Variable -> LambdaExpr
alphaRed (Abstr x body) v = Abstr v (impl x v body [])
    where impl x v (Var z) xs = subst x v z xs
          impl x v (Appl e1 e2) xs = Appl (impl x v e1 xs) (impl x v e2 xs)
          impl x v (Abstr z b) xs = Abstr z (impl x v b (z:xs))
          subst x v z xs
            | z == x = if not $ elem v xs then Var v
                      else error "Invalid substitution"
            | otherwise = Var z

-- provede danou beta redukci
betaRed :: LambdaExpr -> LambdaExpr
betaRed (Appl (Abstr var body) expr) = impl var body expr []
    where impl v (Var x) e xs = subst v x e xs
          impl v (Appl e1 e2) e xs = Appl (impl v e1 e xs) (impl v e2 e xs)
          impl v (Abstr x b) e xs = Abstr x (impl v b e (x:xs))
          subst v x e xs
            | x == v && (not $ elem x xs) = if xs\\(unboundVars e) == xs then e
                                             else error "Invalid substitution"
            | otherwise = Var x

-- provede na daném výrazu eta redukci (pokud to lze udelat)
etaRed :: LambdaExpr -> LambdaExpr
etaRed origExp@(Abstr x (Appl e (Var z)))
    | x == z && (not $ elem x (unboundVars e)) = e
    | otherwise = origExp

```


3 Syntaxe ve funkcích

3.1 Unifikace vzorů

Vzory tvoří určitá schémata, kterým mohou data odpovídat (pattern matching). Vzory se ověřují shora dolů. Obecné vzory se většinou píší nakonec. Absence obecného vzoru může způsobit chybu při zadání neočekávaného vstupu. Anonymní proměnné (nevyužité) se značí znakem `_`.

3.1.1 Konstanta jako vzor

```
lucky :: (Integral a) => a -> Bool
lucky 7 = True
lucky x = False

sayMe :: (Integral a) => a -> String
sayMe 1 = "Jedna!"
sayMe 2 = "Dva!"
sayMe 3 = "Tri!"
sayMe x = "Není mezi 1 a 3."

sgn :: (Num t, Ord t, Num u) => t -> u
sgn 0 = 0
sgn x = if x>0
        then 1
        else (-1)

factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

3.1.2 N-tice jako vzor

```
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)

addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)      -- bez vyuziti vzoru
addVectors' (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)  -- s vyuzitim vzoru

first :: (a, b, c) -> a
first (x, _, _) = x
second :: (a, b, c) -> b
second (_, y, _) = y
third :: (a, b, c) -> c
third (_, _, z) = z

sumTuple :: (Num t) => [(t, t)] -> [t]
sumTuple l = [a+b | (a,b) <- l]                      -- sumTuple [(4,3),(2,4) ,(3,1)]
```

3.1.3 Seznam jako vzor

```
len :: (Num u) => [t] -> u
len [] = 0
len (_:xs) = 1 + len xs

head' :: [a] -> a
head' [] = error "Error!"
head' (x:_) = x

tell :: (Show a) => [a] -> String
tell [] = "Seznam je prazdny."
tell (x:[]) = "Seznam obsahuje 1 prvek: " ++ show x
tell (x:y:[]) = "Seznam obsahuje 2 prvky: " ++ show x ++ " a " ++ show y
tell (x:y:_) = "Seznam je dlouhy. Prvni 2 prvky jsou: " ++ show x ++ " a " ++ show y

sumlist :: (Num a) => [a] -> a
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs
```

3.1.4 Zástupný vzor

Zástupný vzor se značí znakem **@**.

```
capital :: String -> String
capital "" = "Prazdny!"
capital all@(x:xs) = "Prvni pismeno retezce " ++ all ++ " je " ++ [x]
```

3.1.5 Vzor n+k

```
factorial :: (Integral t) => t -> t
factorial 0 = 1
factorial (n+1) = (n+1) * factorial n
```

3.2 Strážné podmínky

Stráže jsou obdobou *if then else* podmínky, ale jsou čitelnější. Podmínka je booleanový výraz. Pokud nepoužijeme výraz *otherwise* (jeho hodnota je vždy *True*), vyhodnocení přejde na následující vzor, pokud žádná strážná podmínka nevyhovuje.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | weight / height ^ 2 <= 18.5 = "Podvyziva"
  | weight / height ^ 2 <= 25.0 = "Normal"
  | weight / height ^ 2 <= 30.0 = "Obezita"
  | otherwise                  = "Sumo"
```

```
max' :: (Ord a) => a -> a -> a
max' a b
  | a > b      = a
  | otherwise = b
```

```
sgn :: (Num a, Ord a, Num a1) => a -> a1
sgn n
  | n < 0 = -1
  | n > 0 = 1
  | otherwise = 0
```

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n
  | n < 0 = error "Error!"
  | otherwise = n * factorial (n-1)
```

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
  | a > b      = GT
  | a == b     = EQ
  | otherwise = LT
```

3.3 Lokální definice

Lokální definice se provádí pomocí klíčového slova *where*. V definicích lze používat vzory. Definovat můžeme konstanty, ale také funkce. Bloky kódu je nutné zarovnávat pod sebe. Konstrukce *where* se mohou větvit.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= skinny = "Podvyziva"
  | bmi <= normal = "Normal"
  | bmi <= fat    = "Obezita"
  | otherwise     = "Sumo"
  where heightsqr = height ^ 2
        bmi      = weight / heightsqr
        skinny   = 18.5
        normal   = 25.0
        fat      = 30.0

-- where s použitím vzoru
...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)

initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
  where (f:_) = firstname
        (l:_) = lastname

-- where s definicí funkce
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
  where bmi weight height = weight / height ^ 2

-- vypočet ordinalní hodnoty daného znaku
ord' :: Char -> Int
ord' x = impl x 0
  where impl x i
        | ['\0' ..]!!i == x = i
        | otherwise         = impl x (i + 1)

-- převod ordinalní hodnoty na znak s ošetřením vyjimky
chr' :: Int -> Maybe Char
chr' i
  | i >= 0    = Just $ ['\0' ..]!!i
  | otherwise = Nothing

-- vypsání všech prvočísel pomocí Erastanova síta
primes = 2:[ x | x <- [3..], test x primes ]
  where test x (p:ps) | x `mod` p == 0 = False
                     | x < p*p = True
                     | otherwise = test x ps
```

Další možnost je definice pomocí klíčových slov *let in*. Umožňuje navázání proměnné kdekoliv ve funkci, ale nelze ji použít ve strážných podmínkách. Lze je rovněž použít pro ověřování vzorů. Konstrukce je sama o sobě výraz, kdežto *where* je jen syntaktický konstrukt. Konstrukce *let* se tedy může vyskytovat stejně jako *if* téměř kdekoliv. Pro oddělení více proměnných se používá středník *;*. Konstrukce *let* se také používá pro definice v interaktivním režimu GHCi. V tomto případě se nepoužívá *in*.

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r^2
    in sideArea + 2 * topArea

numb = 4 * (let a = 9 in a + 1) + 2

-- funkce s lokální působností
sq = [let square x = x * x in (square 5, square 3, square 2)]

-- oddělení středníky
fb = (let a = 10; b = 20; c = 30 in a*b*c, let foo="Hej "; bar = "ty!" in foo ++ bar)

-- použití vzoru
pt = (let (a,b,c) = (1,2,3) in a+b+c) * 100

-- použití v generatoru seznamu, definici nelze aplikovat v části za |, in se nepíše
calcBmis' :: (RealFloat a) => [(a, a)] -> [a]
calcBmis' xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]

calcBmisFat :: (RealFloat a) => [(a, a)] -> [a]
calcBmisFat xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

3.4 Podmíněný výraz case

Case výraz je syntaktický cukr k ověřování vzorů (je ekvivalentní). Case výrazy lze ovšem použít téměř všude. Je možné je vnořovat.

```
head' :: [a] -> a
head' xs = case xs of []      -> error "Error!"
                  (x:_) -> x

describeList :: [a] -> String
describeList xs = "Seznam je " ++ case xs of []  -> "prazdny."
                                             [x] -> "jednoduchovy."
                                             xs  -> "viceprvkovy."

describeList' :: [a] -> String
describeList' xs = "Seznam je " ++ what xs
    where what []  = "prazdny."
          what [x] = "jednoduchovy."
          what xs  = "viceprvkovy."

len list =
    case list of
        [] -> 0
        _:xs -> 1 + len xs
```

4 Rekurze

Princip rekurze spočívá v definici okrajového případu (který zastaví zanořování, obvykle to bývá identita, např. pro seznam je to prázdný seznam), kde se nedá rekurze aplikovat a funkce, která dělá něco s nějakým prvkem a funkcí aplikovanou na zbytek.

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "Error!"
maximum' [x] = x
maximum' (x:xs)
  | x > maxTail = x
  | otherwise = maxTail
  where maxTail = maximum' xs

maximum'' :: (Ord a) => [a] -> a
maximum'' [] = error "Error!"
maximum'' [x] = x
maximum'' (x:xs) = max x (maximum'' xs)

replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
  | n <= 0 = []
  | otherwise = x:replicate' (n-1) x

take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
  | n <= 0 = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs

reverse' :: [a] -> [a] -- casova slozitest O(n^2)
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]

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

zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys

elem' :: (Eq a) => a -> [a] -> Bool
elem' _ [] = False
elem' a (x:xs)
  | a == x = True
  | otherwise = a `elem'` xs

concatOrdered :: (Ord a) => [a] -> [a] -> [a]
concatOrdered xs [] = xs
concatOrdered [] ys = ys
concatOrdered xxx@(x:xs) yyy@(y:ys)
  | x < y = x : concatOrdered xs yyy
  | otherwise = y : concatOrdered xxx ys

order :: (Ord a) => [a] -> [a]
order [] = []
order (x:xs) = concatOrdered [x] (order xs)
```

```

-- prohození dvojic prvku v seznamu
flipValues :: [a] -> [a]
flipValues [] = []
flipValues (x1:x2:xs) = x2:x1:(flipValues xs)
flipValues (x:[]) = [x]

-- nejvetsi spolecny delitel
gcd' :: Int -> Int -> Int
gcd' 0 y = y
gcd' x 0 = x
gcd' x y = gcd' (y `mod` x) x

-- quicksort
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted

```

4.1 Dopředná a zpětná rekurze

Dopředná rekurze má rekurzivní volání jako poslední část výpočtu. Zpětná rekurze po návratu z rekurzivního volání ještě něco počítá. Lineární rekurze obsahuje jen 1 rekurzivní volání a lze převést na cyklus. Koncová rekurze je dopředně rekurzivní a lineární.

```

-- dopredna rekurze
fib :: (Num t, Ord t, Num t1) => t -> t1
fib 0 = 0
fib 1 = 1
fib n
    | n < 0 = error "Error!"
    | otherwise = fib (n-2) + fib (n-1)

-- koncova rekurze
fib' :: Int -> Int
fib' n = if n < 0
    then error "Error!"
    else f 0 1 n
    where f a _ 0 = a
          f a b n = f b (a+b) (n-1)

reverse'' :: [a] -> [a]
reverse'' xs = rev xs []
    where rev [] ys = ys
          rev (x:xs) ys = rev xs (x:ys)

-- funkce pm [a,b,c,d,e,f] ~> a-b+c-d+e-f
-- f $! g x zajisti, ze se pred aplikaci funkce f spocita hodnota g x
pm list = pm' list 0 where
    pm' [] acc = acc
    pm' [x] acc = x + acc
    pm' (x:y:rest) acc = pm' rest $! (x-y+acc)

```

5 Funkce vyššího řádu

Symbol `$` je aplikace funkce. Má nejvyšší prioritu. Používá se pro ušetření závorek.

```
sqrt (3 + 4 + 9)
sqrt $ 3 + 4 + 9
```

5.1 Částečná aplikace

Funkci není nutné v aplikaci saturovat (dodat ji tolik parametrů, kolik je schopna spotřebovat).

```
add x y = x+y
inc x = add 1 x
inc' = add 1
inc'' = (+) 1
inc''' = (1+)
```

5.2 Curryfikace

Každá funkce v Haskellu má oficiálně jen 1 parametr. Funkce s více parametry jsou curryfikované. Každá funkce bere 1 parametr a může vrátit hodnotu nebo funkci. Využívá se částečné aplikace.

```
multThree :: (Num a) => a -> (a -> (a -> a))
multThree x y z = x * y * z

-- funkce curry a uncurry
uncurry :: (a -> b -> c) -> (a,b) -> c
curry :: ((a,b) -> c) -> a -> b -> c

-- rozdily dvojic v seznamu
rozdily :: [(Integer, Integer)] -> [Integer]
rozdily = map (uncurry (-))
```

5.3 Lambda abstrakce

V Haskellu lze používat zápis jako v lambda kalkulu (anonymní funkce).

```
inc = \x -> x+1
flip' f = \x y -> f y x

-- vytvoreni posloupnosti cisel
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
  | even n = n:chain (n `div` 2)
  | odd n = n:chain (n*3 + 1)

-- pro vsechna cisla mezi 1 a 100, kolik posloupnosti je delsi nez 15?
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
  where isLong xs = length xs > 15

-- pouziti Lambda abstrakce
numLongChains' :: Int
numLongChains' = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```


5.4 Mapy

Aplikace funkce na všechny prvky seznamu. Funkce map lze vyjádřit pomocí generátoru seznamu:

$[f\ x \mid x \leftarrow xs] \sim \text{map } f\ xs$.

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

-- bez vyuziti map
squareAll :: [Int] -> [Int]
squareAll [] = []
squareAll (x:xs) = (x*x) : squareAll xs

lengthAll :: [[a]] -> [Int]
lengthAll [] = []
lengthAll (xs:xss) = (length xs) : lengthAll xss

-- s vyuzitim map
squareAll' xs = map (\x -> x*x) xs
lengthAll' xxs = map length xxs

prependHash :: [[Char]] -> [[Char]]
prependHash xs = map (:) '#' xs

incList :: (Num a) => [a] -> [a]
incList xs = map (+1) xs

-- seznam s fibonacciho radou, 4 zpusoby
fibmap = map fib [0..]
fibgen = [ fib x | x <- [0..] ]
fiblist = 0:1:[ x | x <- zipWith (+) fiblist (tail fiblist) ]
fiblist' = 0:1:[ fiblist'!!x + fiblist'!!(x+1) | x <- [0..] ]

-- squareAll' [1,2,3,4]
-- lengthAll' [[],[1],[1,2]]

-- prependHash ["ahoj", "svete"]

-- incList [1,2,3,4]

-- funkce fib viz kapitola 4.1
```

5.5 Filtry

Filtrování prvků seznamu na základě vyhodnocení booleanovské funkce. Funkce filter lze vyjádřit pomocí generátoru seznamu: $[x \mid x \leftarrow xs, p\ x] \sim \text{filter } p\ xs$.

Dále platí: $[f\ x \mid x \leftarrow \text{list}, g\ x] \sim \text{map } f\ \$ \text{filter } g\ \text{list}$

```
-- definice filtru
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x = x:xs'
  | otherwise = xs'
  where xs' = filter p xs

getOdd xs = filter odd xs
getLessThen x xs = filter (<x) xs

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = quicksort (filter (<x) xs) ++ (x : quicksort (filter (>=x) xs))
```

5.6 Akumulační funkce fold

Fold se často používá pro průchod seznamu. Foldl akumuluje hodnoty z levé strany, foldr z pravé. Foldl1 předpokládá počáteční hodnotu jako první prvek v seznamu. Rovnost foldl a foldr platí pro monoid.

```
-- definice foldr, akumuluje hodnoty z prave strany
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

-- definice foldl, akumuluje hodnoty z leve strany
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

-- priklady pro soucet, soucin a logicky and prvku seznamu
suml = foldl (+) 0
sumr = foldr (+) 0
suml' = foldl1 (+)
sumr' = foldr1 (+)
prodl = foldl (*) 1
prodr = foldr (*) 1
andl (x:xs) = foldl (&&) x xs
andr (x:xs) = foldr (&&) x xs

elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
product' :: (Num a) => [a] -> a
product' = foldr1 (*)
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
head' :: [a] -> a
head' = foldr1 (\x _ -> x)
last' :: [a] -> a
last' = foldl1 (\_ x -> x)

-- bez vyuziti foldr
sumlist :: (Num a) => [a] -> a
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs

spoj :: [t] -> [t] -> [t]
spoj [] ys = ys
spoj (x:xs) ys = x:spoj xs ys

propojlist :: [[a]] -> [a]
propojlist [] = []
propojlist (xs:xss) = spoj xs (propojlist xss)

-- s vyuzitim foldr
sumlist' xs = foldr (+) 0 xs

spoj' lx [] = lx
spoj' lx ly = foldr (:) ly lx

propojlist' xss = foldr (++) [] xss
```

5.7 Skládání funkcí

Kompozice funkcí je asociativní zprava. Výraz $f(g(z\ x))$ je stejný jako $(f \cdot g \cdot z)\ x$.

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

```
((*2) . (+3)) 1 -- 8
```

```
map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
```

```
map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
map (negate . sum . tail) [[1..5],[3..6],[1..7]]
```

6 Monády

Monády umožňují řízení toku programu. Monáda je model výpočtu. Funkce *return* umožňuje zabalení do monády (předstíračka výpočtu). Příkaz *bind* *>>=* zajišťuje navázání dvou výpočtů (vybalení výsledku prvního výpočtu a předání druhému výpočtu). Základní monadické třídy: *Functor*, *Monad*, *MonadZero*, *MonadPlus*.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b      -- m >>= return    =    m
  (>>)  :: m a -> m b -> m b              -- m >> k      =    m >>= (\_ -> k)
  return :: a -> m a                      -- return a >>= k    =    k a

-- modelovani vypoctu, který se nemusí povest (vyjimka)
data Maybe a
  = Nothing -- nepovedlo se
  | Just a

-- databaze otce, deda a pradede
otec :: String -> Maybe String
otec "Karel" = Just "Evzen"
otec "Evzen" = Just "Dobromil"
otec "Dobromil" = Just "Franta"
otec _ = Nothing

otcuvOtec :: String -> Maybe String
otcuvOtec x =
  case otec x of
    Nothing -> Nothing
    Just y -> otec y

otcovaOtceOtec :: String -> Maybe String
otcovaOtceOtec x =
  case otec x of
    Nothing -> Nothing
    Just y ->
      case otec y of
        Nothing -> Nothing
        Just z -> otec z

-- deda a pradede pomoci bind
otcuvOtec' x = otec x >>= otec
otcovaOtceOtec' x = otec x >>= otec >>= otec
```

6.1 IO-monáda **limonáda**

Sekvenční chování zajišťuje příkaz *do*, který sdružuje IO akce. Šipka *<-* zajišťuje navázání na proměnnou (rozbalení).

```
do
  a <- m1
  m2 a
  b <- m3 a
  c <- m4 b
  return (f c b)

-- alternativa k příkazu do pomoci bind
m1 >>= (\ a -> m2 a >>= (\ _ -> m3 a
  >>= (\ b -> m4 b >>= (\ c -> return (f c b))))))

-- alternativa k příkazu do pomoci bind
m1 >>= \ a ->
m2 a >>= \ _ ->
m3 a >>= \ b ->
m4 b >>= \ c ->
return (f c b)
```

```

import IO

getChar :: IO Char           -- nacteni znaku
getLine :: IO String        -- nacteni retezce
putChar :: Char -> IO ()    -- tisk znaku
putStr  :: String -> IO ()  -- tisk retezce
putStrLn :: String -> IO () -- tisk retezce se zalomenim
print   :: (Show a) => a -> IO () -- tisk vseh typu instance Show
sequence [putChar 'a', putChar 'b'] -- seznam akci

-- prace s textem
lines :: String -> [String] -- z retezce seznam radku
unlines :: [String] -> String -- ze seznamu radku retezec
words :: String -> [String] -- z retezce seznam slov
unwords :: [String] -> String -- ze seznamu slov retezec

-- prace se soubory
type FilePath = [Char]
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
hIsEOF :: Handle -> IO Bool
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()

-- zabaleni retezce do monady pomoci return a rozbaleni pomoci sipky do promenne
main = do
    a <- return "ahojky"

-- opakujici obracena slova
main :: IO () -- main je akce (hlavni program), ktera nic nevraci
main = do
    line <- getLine
    if line == ""
        then return () -- zabaleni do monady, protoze funkce je typu IO
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words

-- cteni radku textu
getLine :: IO String
getLine = do x <- getChar
    if x == '\n' then return ""
    else do xs <- getLine
        return (x:xs)

-- vypsani retezce
putStr :: String -> IO ()
putStr str = sequence (map putChar str)

```

```
-- nacteni souboru
import IO
do
  handle <- openFile "C:\\x.y" ReadMode
  cont <- hGetContents handle
  putStr cont
  hClose handle
```

```
-- nacteni souboru a vypis jednotlivych slov na radky
changeText txt = stream
  where lns = lines txt
        wrds = map words lns
        newlines = map unlines wrds
        stream = unlines newlines

testopen filename = catch
  (openFile filename ReadMode)
  (\_ -> error ("Bad file: " ++ filename) )

processFile filename = do
  handle <- testopen filename
  cont <- hGetContents handle
  putStr (changeText cont)
  hClose handle
```

```
-- zobrazi z prvnioho souboru radky, ktere jsou i ve druhem souboru v puvodnim poradi
copyOut f1 f2 = do
  h1 <- openFile f1 ReadMode
  h2 <- openFile f2 ReadMode
  c1 <- hGetContents h1
  c2 <- hGetContents h2
  putStr $ unlines $ [l1 | l1 <- lines c1, l2 <- lines c2, l1==l2]
  hClose h2
  hClose h1
```

```
-- pocet radku v souboru
countLines file = do
  hf <- openFile file ReadMode
  a <- countLine 1 hf
  putStr $ (show a) ++ "\n"
  hClose hf
  where countLine number hf = do
    input <- hGetLine hf
    end <- hIsEOF hf
    if end
      then return number
      else countLine (number+1) hf

-- pocet slov na prvnich N radcich
countWordsN file n = do
  contents <- readFile file
  return $ length $ words $ unlines $ take n $ lines contents
```

```
-- odchyceni vyjimky
catch :: IO a -> (IOError -> IO a) -> IO a
getc = catch getChar (\e -> if isEOFError e then return '\n' else ioError e)
```