PENOPORA:
Haskell výpisky

Zdroje: haskell.cz, opora a cvičení z předmětu FPR @ FIT BUT

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

```
-- radkovy komentar
{- blokovy komentar -}
```

1.4 Základní aritmetika

```
2+5               -- scitani
49*100            -- nasobeni
255-124           -- odcitani
5/2               -- deleni
10^2              -- druha mocnina
3*(-5)            -- zaporne cislo v zavorkach
50 * (100 - 4999) -- slozeny vyraz
True && False     -- logicke and
True || False     -- logicke or
not True          -- negace
5==5              -- rovnost
5/=5              -- nerovnost
```

1.5 Volání funkcí

```
succ 6            -- vraci naslednika, argumenty oddeleny mezerou
pred 6            -- vraci predchudce
min 100 10        -- vraci mensi ze dvou prvku
max "ahoj" "cau"  -- vraci vetsi ze dvou prvku
succ 9 * 10       -- vraci naslednika cisla 9 a pote se nasobi 9, tedy 100
succ (9 * 10)     -- vraci naslednika soucinu 9*10, tedy 91
div 92 10         -- prefixovy zapis
92 `div` 10       -- infixovy zapis prefixoveho
(+) 2 5           -- prefixovy zapis infixoveho
odd 1             -- vraci True pokud je cislo liche
even 2            -- vraci True pokud je cislo sude
error "Chyba!"    -- vyvolani vyjimky
```

1.6 Definice funkcí

Názvy proměnných a funkcí začínají malým písmem. V názvu funkcí se může vyskytovat znak apostrofu '. Argumenty se oddělují mezerníkem. 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 vlastni funkce v souboru
doubleUs x y = doubleMe x + doubleMe y    -- definice vlastni funkce v souboru
conanO'Brien = "Ahoj, ja jsem Brian!"     -- definice (pojmenovani) v souboru
```

1.7 Podmínky

Podmínky se mohou vyskytovat téměř kdekoliv. *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]                       -- spojeni seznamu
"nazdar " ++ "chlape"                    -- spojeni retezcu
1:[2,3]                                  -- pridani prvku na zacatek seznamu
1:2:3:[]                                 -- pridani vice prvku do seznamu
'a':"hoj"                                -- pridani znaku na zacatek retezce
[4,8,15,16,23,42] !! 0                   -- ziskani prvniho prvku seznamu
"nazdar" !! 2                            -- ziskani tretiho znaku retezce

let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]    -- definice vnoreneho seznamu
b ++ [[1,1,1,1]]                                     -- spojeni vnorenych seznamu
[6,6,6]:b                                            -- pridani seznamu na zacatek
b !! 2                                               -- ziskani tretiho prvku

head [4,8,15,16,23,42]             -- vrati prvni prvek seznamu
tail [4,8,15,16,23,42]             -- vrati vse mimo prvni prvek seznamu
last [4,8,15,16,23,42]             -- vrati posledni prvek seznamu
init [4,8,15,16,23,42]             -- vrati vse mimo posledni prvek seznamu
length [4,8,15,16,23,42]           -- vrati delku seznamu
null [4,8,15,16,23,42]             -- vrati True pokud je seznam prazdny
reverse [4,8,15,16,23,42]          -- obrati seznam
take 3 [4,8,15,16,23,42]           -- vybere dany pocet prvku ze zacatku seznamu
drop 3 [4,8,15,16,23,42]           -- zahodi dany pocet prvku a vrati zbytek seznamu
minimum [4,8,15,16,23,42]          -- vrati nejmensi prvek
maximum [4,8,15,16,23,42]          -- vrati nejvetsi prvek
sum [4,8,15,16,23,42]              -- vrati soucet vsech prvku seznamu
product [4,8,15,16,23,42]          -- vrati soucin vsech prvku seznamu
elem 8 [4,8,15,16,23,42]           -- vrati True pokud je dany prvek v seznamu

['a'..'z']                         -- rozsah znaku
[1..20]                            -- rozsah cisel
[2,4..20]                          -- rozsah cisel s prirustkem
[20,19..1]                         -- klesajici rozsah cisel
[1..]                              -- nekonecny seznam
take 10 [0,2..]                    -- prvnich 10 nasobku cisla 2
cycle [1,2,3]                      -- cykleni seznamu do nekonecna
repeat 5                           -- cykleni prvku do nekonecna
take 12 (cycle [1,2,3])            -- prvnich 12 prvku nekonecneho seznamu
replicate 3 10                     -- vrati tri vyskyty cisla 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, ktere nejsou povolene v nazvu identifikatoru
let identifStr xs = [x | x <- xs, elem x ['A'..'z'] || elem x ['0'..'9'] || x == '_']

-- fitrovani vnorenych 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

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
```

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

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

2.7 Typová synonyma

```
-- typova synonyma
type String = [Char]
type ComplexF = (Float, Float)                  -- pojmenovani slozeneho typu
type Matrix a = [[a]]                           -- parametrizace typu

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

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

Na rozdíl od typových synonym není nutné v programu explicitně přidávat typovou signaturu. Derivace pomocí klíčového slova *deriving* lze užít u typových tříd Eq, Ord, Ix, 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

Parametry typu a klauzule *deriving* jsou nepovinné.

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

2.9.1 Výčtové typy

```
-- 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 rozsirenym datovym konstruktorem Grayscale
data Color' = Red' | Green' | Blue' | Grayscale Int

getLevelOfGray (Grayscale n) = n
getLevelOfGray _ = 0

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

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

2.9.3 Parametrické typy

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

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

dotProd :: (Num a) => Vector a -> Vector a -> a     -- skalarni soucin
dotProd (Vec l1 v1) (Vec l2 v2) =
      if l1 /= l2
      then error "Bad size!"
      else foldl1 (+) $ zipWith (*) v1 v2

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

2.9.4 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 (==).

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

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

-- pouziti 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)
```

3 Syntaxe ve funkcích

3.1 Unifikace vzorů

Vzory tvoří určitá schémata, kterým mohou data odpovídat. 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 = "Neni 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 booleanovský 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 pouzitim 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 definici funkce
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
     where bmi weight height = weight / height ^ 2
```

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 se 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 lokalni pusobnosti
sq = [let square x = x * x in (square 5, square 3, square 2)]

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

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

-- pouziti v generatoru seznamu, definici nelze aplikovat v casti za |, in se nepise
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.

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

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

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

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]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]

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

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)
```

5 Funkce vyššího řádu

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 vracet 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.

```
inc = \x -> x+1
```

5.4 Mapy

Aplikace funkce na všechny prvky seznamu. Funkce map lze vyjádřit pomocí generátoru seznamu:
[f x | x <- xs] ~ 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                  -- squareAll' [1,2,3,4]
lengthAll' xxs = map length xxs                     -- lengthAll' [[],[1],[1,2]]

prependHash :: [[Char]] -> [[Char]]
prependHash xs = map ((:) '#') xs                   -- prependHash ["ahoj", "svete"]

incList :: (Num a) => [a] -> [a]
incList xs = map (+1) xs                            -- incList [1,2,3,4]

-- seznam s fibonacciho radou, 4 zpusoby
fibmap = map fib [0..]                              -- funkce fib viz kapitola 4.1
fibgen = [ fib x | x <- [0..] ]
fiblist = 0:1:[ x | x <- zipWith (+) fiblist(tail fiblist) ]
fiblist' = 0:1:[ fiblist'!!x + fiblist'!!(x+1) | x <- [0..] ]
```

5.5 Filtry

Filtrování prvků seznamu na základě vyhodnocení booleanovské funkce. Funkce filter lze vyjádřit pomocí generátoru seznamu: [x | x <- xs, p x] ~ filter p xs.
Dále platí: [f x | x ← list, g x] ~ map f $ filter g 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. Foldx1 předpokládají počáteční hodnotu jako první prvek v seznamu.

```
-- 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

-- 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
```

6 IO

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

```
import IO

getChar :: IO Char                          -- nacteni znaku
putStr :: String -> IO ()                   -- tisk řetězce
sequence [putChar 'a', putChar 'b']         -- seznam akcí

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

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

-- 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
```

```
-- nacteni souboru a vypis jednotlivych slov na radky
import IO
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
```

7 Další příklady

```
-- vypis vsech prvocisel pomoci Erastenova sita
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
```

```
-- datovy typ pro index-sekvencni vyhledavaci strom
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)
```

```

type Variable = Char
data LambdaExpr =
      Var Variable
      | Appl LambdaExpr LambdaExpr
      | Abstr Variable LambdaExpr
            deriving (Show, Eq)

data Lexsym = LPar | RPar | Dot |
      Id Variable | Lambda
            deriving (Show, Eq)
```