Užitečné drobnosti

```
error::String->a
Date.Trace.trace::String->a->a
(.)::(b->c)->(a->b)->(a->c)       rev = a . b . c    místo rev x = a (b (c (x)))
($)::(a->b)->a->b                 rev x = a $ b $ c x místo rev x = a (b (c (x)))
infix priorita operátor           priority 0–9, infixl a infixr pro asociativní; default infixl 9

f .> g = g . f
x $> f = f $ x
infixl 0 $>
```

Líné a striktní vyhodnocování

```
seq::a->b->b
```

Primitivum jazyka, při svém vyhodnocení vyhodnotí svůj první parametr, pak druhý a ten vrátí.

```
data Complex = !Float :+ !Float          Oba Floaty jsou striktní, tj. vždy vyhodnocené

...let !a=1`div`0 in...                  Rozšíření GHC; do a se ihned dosadí výsledek výpočtu
 ->let  a=1`div`0 in a`seq`...
```

Poznámka: všimněte si toho `:+` . Je to datový kontruktor ve formě operátoru – ty musí začínat dvojtečkou.

Datové typy

♣ Algebraické datové typy

```
data Tree a = Node a (Tree a) (Tree a) | Nil
data RoseTree a = RoseTree a [RoseTree a]
data Color a = RGB a a a | HSV a a a | Gray a
```

♣ Typová synonyma

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

♣ Newtype

```
newtype Parser a = Parser (String -> (String, a))
```

♣ Recordy

```
data S = S {znaku, slov, radek::Int, slovo::Bool}  Jako S Int Int Int Bool
newStav = S {slov=4, slovo=False, radek=2, znaku=1}
addWord s = let a=slov s in s {slov = a+1}         Nebo rovnou s{slov=1+slov s}
data S = A {slov::Int} | B Int | C {slov::Int}     Korektní, slov::S->Int
data S = A {slov::Int} | B {slov::Float}           Nejde!
data S = S {slov::Int}; slov::cokoliv              Nejde, slov nesmí být znovu deklarována
```

Typové třídy

```
elem::a->[a]->Bool
elem _ []     = False
elem a (x:xs) = if a==x then True else elem a xs        Kde se vezme == ?

class Eq a where
  (==),(/=)::a->a->Bool
instance Eq Bool where
   True==True = True;  False==False = True;   _==_ = False
   True/=True = False; False/=False = False;  _/=_ = True

class Eq a where
  (==),(/=)::a->a->Bool
  (==) = not . (/=)
  (/=) = not . (==)
elem::(Eq a)=>a->[a]->Bool

class Eq a=>Ord a where ...
```

♣ Základní typové třídy: *Eq, Ord, Read, Show, Enum, Bounded*
♣ Číselné typové třídy: *Num, Integral, Fractional, Real, Floating, RealFrac, RealFloat*
♣ Další užitečné: *Bits, Random, Ix, IArray, MArray, Functor, Monad, MonadPlus, ...*
Parametrizovat se dá přes libovolný typ. A typ je například také `a->b` s typovým konstruktorem `(->)`.

```
data Tree a = E | V a [Tree a] deriving (Eq, Ord, Show, Read)
data Colour = Red | Green | Blue deriving (Eq, Ord, Show, Read, Enum, Bounded)
data Pair a = P a a deriving (Eq, Ord, Show, Read, Bounded)
```

Typy polymorfních metod

```
apply f = (f 3, f True)
apply id

let f = id in (f 3, f True)
```

Moduly

```
module Test (f) where

import B
import C (func)
import C hiding (func)
import qualified D
import qualified E as EEE

f x = func x
g x = ...
```

♣ Hierarchické moduly

```
module Data.Array
module Control.Monad
```

Standardní pole

```
array     :: (Ix a)=>(a,a)->[(a,b)]->Array a b   array (1,10) [(i,i)|i<-[1..10]]
listArray :: (Ix a)=>(a,a)->[b]->Array a b       listArray (1,10) [1..10]
(!)       :: (Ix a)=>Array a b->a->b             a!1
bounds    :: (Ix a)=>Array a b -> (a,a)  indices::(Ix a)=>Array a b->[a]
elems     :: (Ix a)=>Array a b -> [b]    assocs ::(Ix a)=>Array a b->[(a,b)]
(//)      :: (Ix a)=>Array a b->[(a,b)]->Array a b a//[(1,2), (3,4)], dělá celou kopii
```

Pole jsou líná v hodnotách – nevyhodnocují, dokud nemusí.
Vícerozměrná pole pomocí `array ((1,1),(100,100)) [((i,j),i+j)|i<-[1..100],j<-[1..100]]`

Používání kompilátoru a interpreteru

♣ Kompilace: `ghc --make f.hs`, `-O2` optimalizace, `-fglasgow-exts` různá rozšíření. Musí obsahovat `main`.
♣ Interpreter: `ghci f.hs` nebo `hugs f.hs`, `hugs`u můžete dát `-98` místo `-fglasgow-exts`
`:l f.hs` načte `f.hs`; `:r` reloadne načtené moduly; `:t expr` vypíše typ `expr`; ostatní provede daný příkaz
♣ Skripty: vytvořte soubor `.hs`, `chmod a+x`, první řádek `#!/usr/bin/runhaskell` či `runghc` či `runhugs`
♣ Literate: haskell zná i soubory `.lhs`: každá řádka je komentář, pokud nezačíná znakem > nebo není v bloku začínajícím `\begin{document}` a končícím `\end{document}`. Nemixujte to v jednom souboru. Navíc kolem bloku řádek začínajícím > musí být prázdná řádka.

Domácí úkoly

♣ Máte dáno **data** `Tree a = Empty | Tree a [Tree a]`. Napište ručně instanci `Show (Tree a)`, aby fungovala v lineárním čase.

♣ Napište funkci, která vrací nekonečný seznam rostoucích čísel, která nemají jiné prvočíselné dělitele než 2, 3, 5, tj. čísel tvaru 2^i * 3^j * 5^k. Složitost získání prvních K prvků musí být O(K).

♣ Napište funkci, která pro dva dané seznamy najde délku jejich nejdelšího společného ne nutně souvislého podseznamu. Časová složitost by měla být nejvíc O(součin délek těchto seznamů).

♣ Napište funkci `sumall`, která dostane libovolný nezáporný počet `Int`ů a vrátí jejich součet.

♣ Napište funkci `printf`, která bude umět vypisovat dané argumenty podle formátovacího řetězce. Podporujte alespoň %c %d a %s. Funkce nemusí fungovat přesně tak, jako její céčková varianta, ale `printf "Ahoj %d %s%c" (5::Int) "člověků" '!'` by mělo fungovat.

♣ Navrhněte funkci `mulAll`, které jako první parametr nějak zadáte počet a ona pak vynásobí tolik parametrů typu `Integer`. Takže typ `mulAll "dva"` musí být `Integer->Integer->Integer`. Tu "dvojku" nemůžete zadat číslem ani stringem, ale nějak jinak (klidně to může být unárně).