Líné vyhodnocování

```
ignore_fst::Int->Int->Int
ignore_fst x y = y
```
Výraz `ignore_fst (1`div`0) 5` nespadne, protože `1`div`0` se nevyhodnotí.

Vyhodnocuje se "až je třeba" – ale kdy je třeba? Buď v pattern–matchingu nebo v **case**.
Například *Int* je v GHC `data Int = I# Int#`. Výraz `x=1`div`0` zůstává uložen jako odkaz, dokud někdo neudělá třeba `case x of (I# x#)...` a v té chvíli dojde k výpočtu hodnoty.

```
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.

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
```

Užitečné drobnosti

where nemůže být úplně všude. Může být jenom za definicí funkce nebo za libovolnou větví **case**.

```
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
```

Domácí úkoly z minula

```
import Text.Printf
wc = interact $ unlines . zipWith (printf"%6d\t%s") [(1::Int)..] . lines
cat_n = interact $ (++"\n") . unwords . map show . res
  where res s = let l=lines s; w=words s
                in [length l, length w, sum $ map length w]
cat_n' = interact $ write . foldl count (0::Int,0::Int,0::Int,1::Int)
  where count (!l,!w,!c,!ls) z | z=='\n'   = (l+1,w,c,1)
                               | z==' '    = (l,w,c,1)
                               | otherwise = (l,w+ls,c+1,0)
        write (l,w,c,_) = printf "%d %d %d\n" l w c

data Tree a = Nil | V a (Tree a) (Tree a) deriving Show
perf n = if n==1 then Nil else let son=perf (n`div`2) in V 1 son son
onepass t = let (sum, num, tree)=onepass' t (sum`div`num) in tree
  where onepass' Nil       _ = (0, 0, Nil)
        onepass' (V x l r) n = let (ls, ln, lt) = onepass' l n
                                   (rs, rn, rt) = onepass' r n
                               in (x+ls+rs, 1+ln+rn, V n lt rt)
l2t ls = snd $ l2t' (length ls) ls
  where l2t' 0 ls = (ls, Nil)
        l2t' n ls = let (l:ls', left)=l2t' (   n `div`2) ls
                        (ls'', right)=l2t' ((n-1)`div`2) ls'
                    in (ls'', V l left right)
```

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)
```

Následují různá rozšíření typových tříd

```
class Collection col elem where
  empty::col elem
  contains::elem->col elem->Bool
```

Funguje dobře, dokud nebudeme chtít mít množinu čísel reprezentovanou v Integeru pomocí bitů.
Pak totiž v typu `empty` nedává `Integer Int` smysl.

```
class Collection colelem elem where
  empty::colelem                          --"Tohle způsobí kompilační chybu při prv
ním použití
  contains::elem->colelem->Bool

class Collection colelem elem | colelem->elem where...
instance Collection [a] a where...   instance Collection Integer Int where...
```

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 f.hs -o f`, `-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 na příště (do úterní půlnoci se mohou na webu objevit ještě další)

♣ Pro dva zadané stringy najděte délku jejich nejdelšího společného podřetězce.
♣ Napište jednoduché derivovátko `::String->String`, derivuje se podle `x`. Musí to umět alespoň
`+ * ^ sin cos log x` a čísla. Operátory musí jít zadávat infixově a musí funovat priority `+ * ^`,
takže `3 + 4 * 5 ^ 2` je `3 + (4 * (5^2))`. Snažte si co nejvíc ulehčit práci pomocí třídy `Read`.
Na následující úložky smíte použít `-fglasgow-exts` (v Hugsu `-98`). Uvědomte si, že `(->)` je typový konstruktor.
♣ Napište funkci `sumall`, která dostane libovolný nezáporný počet `Int`ů a vrátí jejich součet. (Jde bez rozšíření.)
♣ Napište funkci `maxall`, která dostane libovolný kladný počet porovnatelných prvků a vrátí jejich maximum.
Všechny parametry musí být stejného typu, ale ten typ je libovolný porovnatelný.
♣ 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. Beru unární kódování, ale za decimální bude bonus.