

Vyhodnocovací strategie (lenost vs. striktnost)

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

Líné vyhodnocování: příklad

`fst (1, undefined) \rightsquigarrow^* 1`

- druhá složka dvojice se vůbec nevyhodnotí

Líne vyhodnocování: příklad

```
fst (1, undefined)  $\rightsquigarrow^*$  1
```

- druhá složka dvojice se vůbec nevyhodnotí

```
let fact n = product [1..n]
```

```
in map fact [1000000,999999..0] !! 999995  $\rightsquigarrow^*$  120
```

- jednotlivé prvky jsou vypočítány jen když jsou vyžádány

Líné vyhodnocování: příklad

```
fst (1, undefined)  $\rightsquigarrow^*$  1
```

- druhá složka dvojice se vůbec nevyhodnotí

```
let fact n = product [1..n]  
in map fact [1000000,999999..0] !! 999995  $\rightsquigarrow^*$  120
```

- jednotlivé prvky jsou vypočítány jen když jsou vyžádány

```
let fibs = 0:1:zipWith (+) fibs (tail fibs)  
in fibs !! 100  $\rightsquigarrow^*$  354224848179261915075
```

- seznam je konstruován líně, tedy jen prvky, které jsou potřeba

- každá hodnota v Haskellu může být
 - a) nevyhodnocená – funkce počítající hodnotu
 - b) vyhodnocená
- pokud se snažíme nevypočítanou hodnotu přečíst, vypočítá se
- při příštím čtení je již vypočítaná
- vyhodnocuje se vždy jen to co je nezbytně nutné

Líne vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)
```

Líne vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)

facts !! 1
```

Líné vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)

facts !! 1
  ~> (map fact [0..]) !! 1
```


Líne vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
```

Líné vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
  ~> (fact 0 : map fact [1..]) !! 1
```

Líné vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
  ~> (fact 0 : map fact [1..]) !! 1
  ~> (map fact [1..]) !! (1 - 1)
```

Líne vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
  ~> (fact 0 : map fact [1..]) !! 1
  ~> (map fact [1..]) !! (1 - 1)
  ~> (map fact (1:[2..])) !! (1 - 1)
```

Líne vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
  ~> (fact 0 : map fact [1..]) !! 1
  ~> (map fact [1..]) !! (1 - 1)
  ~> (map fact (1:[2..])) !! (1 - 1)
  ~> (fact 1 : map fact [2..]) !! (1 - 1)
```

Líné vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
  ~> (fact 0 : map fact [1..]) !! 1
  ~> (map fact [1..]) !! (1 - 1)
  ~> (map fact (1:[2..])) !! (1 - 1)
  ~> (fact 1 : map fact [2..]) !! (1 - 1)
  ~> (fact 1 : map fact [2..]) !! 0
```

Líne vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]      !! _ = error "(!!): index too large"
(x:_)   !! 0 = x
(_:xs)  !! n = xs !! (n - 1)
```

```
facts !! 1
  ~> (map fact [0..]) !! 1
  ~> (map fact (0:[1..])) !! 1
  ~> (fact 0 : map fact [1..]) !! 1
  ~> (map fact [1..]) !! (1 - 1)
  ~> (map fact (1:[2..])) !! (1 - 1)
  ~> (fact 1 : map fact [2..]) !! (1 - 1)
  ~> (fact 1 : map fact [2..]) !! 0
  ~> fact 1 ~> product [1..1] ~>* 1
```

Striktност a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
      | y == min = ys
      | otherwise = y : withoutMin ys
```


Striktnost a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
      | y == min = ys
      | otherwise = y : withoutMin ys
```

- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po [])
- vyhodnotí všechny prvky až do té míry jak vyžaduje (<)
- ale produkuje výstup líně!

Striktnost a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
      | y == min = ys
      | otherwise = y : withoutMin ys
```

- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po [])
- vyhodnotí všechny prvky až do té míry jak vyžaduje (<)
- ale produkuje výstup líně!
 - jaká je složitost `head (selectsort [10000,9999..0])`?

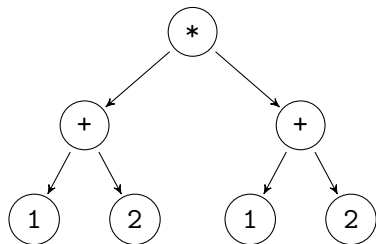
Striktnost a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
      | y == min = ys
      | otherwise = y : withoutMin ys
```

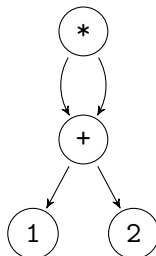
- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po [])
- vyhodnotí všechny prvky až do té míry jak vyžaduje (<)
- ale produkuje výstup líně!
 - jaká je složitost `head (selectsort [10000,9999..0])`?
 - $\mathcal{O}(n)$

Sdílení podvýrazů

$(1 + 2) * (1 + 2)$

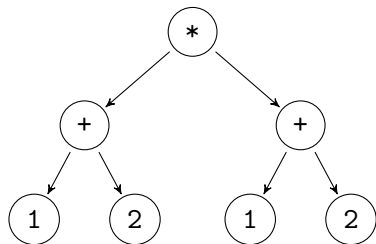


`let a = 1 + 2 in a * a`

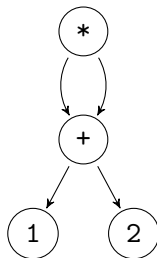


Sdílení podvýrazů

$(1 + 2) * (1 + 2)$



`let a = 1 + 2 in a * a`



`square a = a * a`

`> square (1 + 2)`

- standardně GHCi vypisuje hodnoty pomocí `print`, což obvykle způsobuje plné vyhodnocení
- pomocí `:sprint` lze vypsát bez vynucení vyhodnocení
- pozor, polymorfní výrazy se vyhodnocují vždy znovu

Sdílení v GHCi

- standardně GHCi vypisuje hodnoty pomocí `print`, což obvykle způsobuje plné vyhodnocení
- pomocí `:sprint` lze vypsát bez vynucení vyhodnocení
- pozor, polymorfní výrazy se vyhodnocují vždy znovu

```
>λ= let a = 5 + 2 :: Int
>λ= let xs = [a, 1, 2, a, 3]
>λ= :sprint xs
xs = [_,1,2,_,3]
>λ= head xs
3
>λ= :sprint xs
xs = [7,1,2,7,3]
```

Teoretická vsuvka: vyhodnocovací strategie

Striktní vyhodnocovací strategie

- výrazy vyhodnoceny při přiřazení do proměnných
- argumenty vyhodnoceny před voláním
- pořadí vyhodnocování obvykle do značné míry jasné ze zdrojového kódu
- typické pro imperativní jazyky, ale i mnohé funkcionální (*ML)

Teoretická vsuvka: vyhodnocovací strategie

Striktní vyhodnocovací strategie

- výrazy vyhodnoceny při přiřazení do proměnných
- argumenty vyhodnoceny před voláním
- pořadí vyhodnocování obvykle do značné míry jasné ze zdrojového kódu
- typické pro imperativní jazyky, ale i mnohé funkcionální (*ML)

Normální/ne-striktní vyhodnocovací strategie

- výrazy se vyhodnocují až když jsou potřeba
- umožňuje existenci potenciálně nekonečných datových struktur
- pořadí vyhodnocení méně viditelné
- může mít větší spotřebu paměti (nevyhodnocené výrazy)

Teoretická vsuvka: vyhodnocovací strategie

Striktní vyhodnocovací strategie

- výrazy vyhodnoceny při přiřazení do proměnných
- argumenty vyhodnoceny před voláním
- pořadí vyhodnocování obvykle do značné míry jasné ze zdrojového kódu
- typické pro imperativní jazyky, ale i mnohé funkcionální (*ML)

Normální/ne-striktní vyhodnocovací strategie

- výrazy se vyhodnocují až když jsou potřeba
- umožňuje existenci potenciálně nekonečných datových struktur
- pořadí vyhodnocení méně viditelné
- může mít větší spotřebu paměti (nevyhodnocené výrazy)

Líná vyhodnocovací strategie

- normální + sdílení podvýrazů

Co vynucuje vyhodnocení?

Vzory

f1, f2, f3 :: [a] -> ()

f1 _ = ()

f2 (_:_) = ()

f3 (_:_:[]) = ()

f1 undefined \rightsquigarrow^*

Co vynucuje vyhodnocení?

Vzory

`f1, f2, f3 :: [a] -> ()`

`f1 _ = ()`

`f2 (_:_) = ()`

`f3 (_:_:[]) = ()`

`f1 undefined \rightsquigarrow^* ()`

Co vynucuje vyhodnocení?

Vzory

f1, f2, f3 :: [a] -> ()

f1 _ = ()

f2 (_:_) = ()

f3 (_:_:[]) = ()

f1 undefined \rightsquigarrow^* ()

f2 undefined \rightsquigarrow^*

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined ~>* ()
```

```
f2 undefined ~>* *** Exception: Prelude.undefined
```

Co vynucuje vyhodnocení?

Vzory

f1, f2, f3 :: [a] -> ()

f1 _ = ()

f2 (_:_) = ()

f3 (_:_:[]) = ()

f1 undefined \rightsquigarrow^* ()

f2 undefined \rightsquigarrow^* *** Exception: Prelude.undefined

f1 [] \rightsquigarrow^*

Co vynucuje vyhodnocení?

Vzory

f1, f2, f3 :: [a] -> ()

f1 _ = ()

f2 (_:_) = ()

f3 (_:_:[]) = ()

f1 undefined \rightsquigarrow^* ()

f2 undefined \rightsquigarrow^* *** Exception: Prelude.undefined

f1 [] \rightsquigarrow^* ()

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$ 
```

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$  *** Exception: ...
```

Co vynucuje vyhodnocení?

Vzory

f1, f2, f3 :: [a] -> ()

f1 _ = ()

f2 (_:_) = ()

f3 (_:_:[]) = ()

f1 undefined \rightsquigarrow^* ()

f2 undefined \rightsquigarrow^* *** Exception: Prelude.undefined

f1 [] \rightsquigarrow^* ()

f2 [] \rightsquigarrow^* *** Exception: ...

f2 [undefined,undefined] \rightsquigarrow^*

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$  *** Exception: ...
```

```
f2 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$  *** Exception: ...
```

```
f2 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f3 [undefined,undefined]  $\rightsquigarrow^*$ 
```

Co vynucuje vyhodnocení?

Vzory

f1, f2, f3 :: [a] -> ()

f1 _ = ()

f2 (_:_) = ()

f3 (_:_:[]) = ()

f1 undefined \rightsquigarrow^* ()

f2 undefined \rightsquigarrow^* *** Exception: Prelude.undefined

f1 [] \rightsquigarrow^* ()

f2 [] \rightsquigarrow^* *** Exception: ...

f2 [undefined,undefined] \rightsquigarrow^* ()

f3 [undefined,undefined] \rightsquigarrow^* ()

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$  *** Exception: ...
```

```
f2 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f3 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f2 (undefined:undefined)  $\rightsquigarrow^*$ 
```

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$  *** Exception: ...
```

```
f2 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f3 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f2 (undefined:undefined)  $\rightsquigarrow^*$  ()
```


Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined ~>* ()
```

```
f2 undefined ~>* *** Exception: Prelude.undefined
```

```
f1 [] ~>* ()
```

```
f2 [] ~>* *** Exception: ...
```

```
f2 [undefined,undefined] ~>* ()
```

```
f3 [undefined,undefined] ~>* ()
```

```
f2 (undefined:undefined) ~>* ()
```

```
f3 (undefined:undefined) ~>*
```

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined  $\rightsquigarrow^*$  ()
```

```
f2 undefined  $\rightsquigarrow^*$  *** Exception: Prelude.undefined
```

```
f1 []  $\rightsquigarrow^*$  ()
```

```
f2 []  $\rightsquigarrow^*$  *** Exception: ...
```

```
f2 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f3 [undefined,undefined]  $\rightsquigarrow^*$  ()
```

```
f2 (undefined:undefined)  $\rightsquigarrow^*$  ()
```

```
f3 (undefined:undefined)  $\rightsquigarrow^*$  *** Exception: ...
```

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
```

```
f1 _ = ()
```

```
f2 (_:_) = ()
```

```
f3 (_:_:[]) = ()
```

```
f1 undefined ~>* ()
```

```
f2 undefined ~>* *** Exception: Prelude.undefined
```

```
f1 [] ~>* ()
```

```
f2 [] ~>* *** Exception: ...
```

```
f2 [undefined,undefined] ~>* ()
```

```
f3 [undefined,undefined] ~>* ()
```

```
f2 (undefined:undefined) ~>* ()
```

```
f3 (undefined:undefined) ~>* *** Exception: ...
```

- jen co je nezbytně nutné k rozhodnutí, zda hodnota odpovídá VZORU

Co vynucuje vyhodnocení?

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

- `a `seq` b` zaručuje, že `a` bude vyhodnoceno (= jeden výpočetní krok), vrátí `b`
- k vyhodnocení dojde až když něco vynutí výpočet výrazu `a `seq` b`
- nevyhodnocuje `a` plně:
`(undefined, undefined) `seq` 1 ~>* 1`
`undefined `seq` 1 ~>* *** Exception: ...`

Co vynucuje vyhodnocení?

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

- `a `seq` b` zaručuje, že `a` bude vyhodnoceno (= jeden výpočetní krok), vrátí `b`
- k vyhodnocení dojde až když něco vynutí výpočet výrazu `a `seq` b`
- nevyhodnocuje a plně:
`(undefined, undefined) `seq` 1 ~>* 1`
`undefined `seq` 1 ~>* *** Exception: ...`

Striktní aplikace (`$!`) :: `(a -> b) -> a -> b`

- obdoba (`$`), ale vynutí vyhodnocení argumentu před aplikací
- `f1 $! undefined ~>* *** Exception: ...`

Proč striktnost?

- striktní výpočet může být rychlejší, paměťově efektivnější

```
>λ= :set +s  
(0.18 secs, 74075592 bytes)  
>λ= const () $! product [1..100000]  
( )  
(18.32 secs, 9052106864 bytes)  
>λ= :m + Data.List  
>λ= const () $! foldl' (*) 1 [1..100000]  
( )  
(2.73 secs, 9045968328 bytes)
```

- foldl' je striktní verze foldl

Proč striktnost?

```
readFiles :: [FilePath] -> IO [String]
readFiles paths = mapM readFile paths
```

- readFile vrací obsah souboru líně
- soubor je zavřen až když je dočten na konec!
- lehce můžeme překročit limit na množství otevřených souborů
- seq nepomůže

Control.DeepSeq

Úplné/hluboké vyhodnocení.

```
import Control.DeepSeq
```

```
readFiles :: [FilePath] -> IO [String]
readFiles paths = forM paths $ \p ->
    withFile p ReadMode $ \h -> do
        c <- hGetContents h
        return $!! c
```

- `($!!) :: NFData a => (a -> b) -> a -> b`
plně vyhodnotí argument, pak aplikuje funkci
- `deepseq :: NFData a => a -> b -> b`
- `class NFData a where`
 `rnf :: a -> ()`

plně vyhodnotí argument

Control.DeepSeq

```
import Control.DeepSeq

data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
               deriving (Show, Eq)

instance NFData a => NFData (BinTree a) where
  rnf Empty = ()
  rnf (Node v t1 t2) = rnf v `seq` rnf t1 `seq`
                      rnf t2
```

- GHC má různá rozšíření Haskellu
- rozšíření zapínáme speciálním pragma komentářem na prvním řádku souboru

```
{-# LANGUAGE <rozsireni> #-}
```

nebo na příkazové řádce pomocí `-X<rozšíření>`, z ghci
`:set -X<rozšíření>`

- různá rozšíření syntaxe, návrhy budoucích standardů, apod.
- https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/ghc-language-features.html

BangPatterns

```
{-# LANGUAGE BangPatterns #-}
```

Umožňuje snadno vynutit striktnost u vzoru, `let` výrazu, hodnoty v typu.

```
data Tuple a b = Tuple !a !b
```

```
lazy, strict :: a -> ()
```

```
lazy _ = ()
```

```
strict !_ = ()
```

- taková hodnota je pak **na nejvyšší úrovni** vyhodnocena před voláním funkce/konstruktoru
- v `let` způsobí vyhodnocení příslušného výrazu před přejitím do `in` podvýrazu (striktní `let`)

```
($!) :: (a -> b) -> a -> b
```

```
f $! x = let !ex = x in f ex
```

StrictData

- language pragma for entire file
- as if all fields in data definition *in this module* were defined with ! (from BangPatterns)
- can be overridden by ~

StrictData

- language pragma for entire file
- as if all fields in data definition *in this module* were defined with ! (from BangPatterns)
- can be overridden by ~

Strict

- implies StrictData
- function definitions, **where/let** binding, **case/do** patterns are also strict by default
- nested bindings and top-level bindings are not strict

StrictData

- language pragma for entire file
- as if all fields in data definition *in this module* were defined with ! (from BangPatterns)
- can be overridden by ~

Strict

- implies StrictData
- function definitions, **where/let** binding, **case/do** patterns are also strict by default
- nested bindings and top-level bindings are not strict

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#extension-StrictData

- `Data.List: foldl'`, `foldl1'`
- `Data.Map.Strict` – map with strict values
- pokročilé věci související s monadickými transformery a `ST`

Naprogramujte variantu ohodnocených binárních stromů se striktními hodnotami (`StrictValTree`), ale línou reprezentací stromové struktury, a variantu, která je plně striktní jak ve struktuře tak v hodnotách (`StrictTree`).

- rozmyslete si, pro jaké použití by se mohli jednotlivé varianty hodit a kdy byste naopak použili línou variantu
- rozmyslete si, jaké prostředky z tohoto cvičení chcete použít pro vynucení striktnosti v jednotlivých případech a proč
- pro obě verze stromů implementujte instanci `Functor`