Cvičení 6: Manipulace s funkcemi, typy, opakování

Před šestým cvičením je zapotřebí znát:

- typy základních entit v Haskellu (čísel, řetězců, seznamu, n-tic);
- základní typové třídy (pro čísla, desetinná čísla, porovnatelné a seřaditelné typy, zobrazitelné typy);
- použití operátoru `(.) :: (b -> c) -> (a -> b) -> (a -> c)` pro skládání (unárních) funkcí;
- co je částečná aplikace;
- základní funkce pro manipulace s čísly a seznamy, včetně základních funkcí vyšších řádů nad seznamy;
- definice a používání vlastních datových typů, včetně rekurzivních;
- znalost pojmů hodnotový a typový konstruktor.

Na cvičení si prosím přineste papír a tužku, budou se hodit.

Pan Fešák doporučuje: Na tomto cvičení se vám bude hodit tužka a papír ještě více než obvykle. Cílem cvičení na papír je procvičit si přemýšlení nad jednotlivými koncepty, bez spoléhání se na interpret.

Pro tohle cvičení si potřebujeme zavést následující dva pojmy.

Definice 6.1: totální funkce. Funkce v Haskellu je **totální** právě tehdy, když pro libovolný platný vstup vyprodukuje v konečném čase výstup typu požadovaného její typovou signaturou. Platným vstupem se rozumí takový, který lze vyhodnotit bez pádu či zacyklení.

Například:

- funkce `(||) :: Bool -> Bool -> Bool` je totální, protože pro libovolné vstupy vrátí platnou hodnotu typu `Bool`;
- funkce `head :: [a] -> a` není totální, protože existuje hodnota `[]`, pro niž tato funkce vyhodí chybu, a tedy nevrátí hodnotu;
- níže uvedená funkce `isEven` není totální, protože pro záporná čísla neskončí.

```
isEven :: Integral a => a -> Bool
isEven 0 = True
isEven 1 = False
isEven x = isEven (x - 2)
```

Funkce v Haskellu obecně totální nejsou. To je jeden z rozdílů mezi funkcemi v Haskellu a funkcemi v matematice (a v předmětu Matematické základy informatiky), kde je funkce bez přídomku totální a pokud potřebujeme, můžeme odlišit *parciální funkce*. Z hlediska matematických základů je tedy například výše uvedená `isEven` parciální (pro definiční obor celých čísel, který odpovídá typu `Integer`).

Poznámka: Všimněte si, že pro funkce pracující s nekonečnými strukturami se nám situace komplikuje, protože často nechceme vyžadovat konečný běh (například protože vytváříme nekonečnou strukturu). Při úvahách o totálnosti v tomto předmětu si však vystačíme bez uvažování nekonečných struktur.

Definice 6.2: chováním různé funkce. Funkce `f` a `g` označíme za **chováním různé**, pokud existuje alespoň jeden vstup na němž se jejich výstup liší. Pokud nejsou funkce chováním různé, pak jsou **chováním stejné** (také *sémanticky ekvivalentní*).

Například:

- funkce `and` a `foldr (&&) True` jsou chováním stejné;
- funkce `(||)` a `(&&)` jsou chováním různé, protože existuje vstup (například `True False`), pro který se jejich výstupy liší.

Etudy

Etuda 6.η.1

Určete nejobecnější typy výrazů. Neuvádějte typové třídy, které jsou již implikovány jinými typovými třídami. Můžete použít interpret k zjištění typů knihovních funkcí a závislostí mezi typovými třídami.

a) `not True`
b) `[]`
c) `["", []]`
d) `[True, []]`
e) `[1, 2, 3.14]`
f) `[1, 2, True]`
g) `filter (\x -> x > 5) [1, 2, 4, 8]`
h) `\x -> show (x ^ x)`
i) `\x -> read x + 2`
j) `\x -> (fromIntegral x :: Double)`

Etuda 6.η.2

Určete typy následujících funkcí:

a)
```
swap (x, y) = (y, x)
```
b)
```
swapIf True (x, y) = (y, x)
swapIf _    (x, y) = (x, y)
```
c)
```
sayLength [] = "empty"
sayLength x  = "nonempty"
```
d)
```
aOrX 'a' _ = True
aOrX _   x = x
```

> **Pan Fešák připomíná:** V případě, že chci otypovat funkci definovanou s více definičními rovnicemi, otypuji každou definiční rovnost zvlášť a pak *unifikuji* typy argumentů na odpovídajících pozicích a typy návratových hodnot. Tedy hledám dosazení za proměnné tak, aby oba typy byly stejné, bez toho, abych zbytečně konkretizoval typové proměnné, u kterých to není nutné.
>
> Unifikace typů v Haskellu funguje v podstatě stejně jako unifikace termů v Prologu. Rozdíl je v tom, že Haskell dělá kontrolu sebevýskytu, protože bez ní by mohly vznikat nekonečné typy.

Etuda 6.η.3 Napište všechny *chováním různé* (definice 6.2) totální funkce (definice 6.1) typu `Bool -> Bool`.

Etuda 6.η.4 S použitím standardních funkcí z modulů `Prelude`, `Data.List` a `Data.Char` naprogramujte následující funkce bez použití vlastních pomocných funkcí, explicitní rekurze a případných zakázaných funkcí nebo konstrukcí.

a) `nth :: [a] -> Int -> a`, která vrátí n-tý prvek seznamu (pokud existuje, jinak může libovolně selhat). Bez použití funkce `(!!) :: [a] -> Int -> a`.

```
[1, 4, 16] `nth` 2 ⇝* 16
[1..] `nth` 10 ⇝* 11
```

b) `firstUppercase :: String -> Char`, která vrátí první velké písmeno ze svého vstupu, nebo `'-'`, pokud tam žádné není.

```
firstUppercase "ahoj Světe!" ⇝* 'S'
firstUppercase "ahoj" ⇝* '-'
```

c) `everyNth :: [a] -> Int -> [a]`, která pro daný (potenciálně nekonečný) seznam vrátí seznam obsahující jeho prvky na pozicích dělitelných druhým argumentem.

```
everyNth [1, 2, 3, 4, 5] 2 ⇝* [1, 3, 5]
everyNth "ahoj" 10 ⇝* "a"
everyNth [0..] 10 ≈ [0, 10..]
```

Poznámka: poslední příklad není vyhodnocení, ale ekvivalence ve smyslu, že oba výrazy se vyhodnotí na stejný seznam (v tomto případě nekonečný).

d) Pokud jste to tak už neudělali, naprogramujte `everyNth` i bez použití lambda funkcí. Místo nich použijte skládání funkcí (formální argumenty samotné funkce `everyNth` neodstraňujte, bylo by to velmi náročné).

Etuda 6.η.5 Definujte funkci `binmap :: (a -> a -> b) -> [a] -> [b]`, která je obdobou `map`, ale aplikuje danou funkci vždy na dva po sobě následující prvky tak, že prvek na pozici i ve výsledném seznamu vznikne z prvků i a $i+1$ vstupního seznamu. Výstupní seznam je tedy o jeden prvek kratší (nebo prázdný, pokud byl vstupní seznam prázdný).

```
binmap (,) [1, 2, 3, 4] ⇝* [(1, 2), (2, 3), (3, 4)]
binmap (-) [14, 12, 8, 3, 1] ⇝* [2, 4, 5, 2]
binmap (+) [1] ⇝* []
binmap (+) [] ⇝* []
```

> **Pan Fešák doporučuje:** Kostru úloh k této kapitole najdete připravenou k použití v souboru 06_data.hs v příloze sbírky nebo ve studijních materiálech v ISu.

6.1 Typy a typové třídy

Př. 6.1.1 Určete, jakou aritu mají funkce podle jejich typu. Aritou rozumíme počet argumentů, které musíme funkci dát, aby typ výsledku nebyl funkční typ.

a) `Eq a => a -> a -> a -> Bool`
b) `(a -> Bool) -> ([a] -> Int)`
c) `[Int -> Int -> Int]`
d) `Int -> Integer -> String -> String`
e) `(Int -> Integer) -> String -> String`
f) `Int -> Integer -> (String -> String)`
g) `Int -> Integer -> [String -> String]`
h) `(a -> b -> c) -> b -> a -> c`

Př. 6.1.2 Určete typy následujících funkcí a popište slovně, co funkce dělají.

a)
```
cm _ []       = []
cm x (y : z) = x y ++ cm x z
```

b)
```
mm []     = (maxBound, minBound)
mm (x:xs) = let (a, b) = mm xs in (min a x, max b x)
```

c)
```
c x y = \z -> x (y z)
```

Př. 6.1.3 Bez použití interpretu určete nejobecnější typy následujících výrazů. Neuvádějte typové třídy, které jsou implikovány jinými typovými třídami. Pokud potřebujete, můžete typy základních funkcí zjistit pomocí interpretu či dokumentace.

6-typy

a) `\x y -> map y x`
b) `\x -> flip replicate`
c) `\x y -> take y [fst x, snd x]`
d) `\x y -> map x . filter y`
e) `\x y -> fromIntegral x `div` y`

Př. 6.1.4 Bez použití interpretu určete aritu následujících funkcí. Aritou rozumíme počet argumentů, které musíme funkci dát, aby typ výsledku nebyl funkční typ.

a) `flip map`
b) `(not . null)`
c) `(\f p -> filter p . map f)`
d) `(\x -> x 1 || x 2)`
e) `flip (not .)`
f) `zipWith id`

Př. 6.1.5 Napište všechny *chováním různé* totální funkce typu `(a -> b) -> a -> b` (viz definice 6.2).

Př. 6.1.6 Napište všechny *chováním různé* totální funkce typu `[a] -> a` (viz definice 6.2).

Př. 6.1.7 Napište všechny *chováním různé* totální funkce typu `(a -> Maybe b) -> Maybe a -> Maybe b` (viz definice 6.2).

Př. 6.1.8 Určete typ funkce `f`. Jak se funkce chová na různých vstupech?

```
f x y       True  = if x > 42 then y else []
f _ (_ : s) False = s
f _ _       _     = "IB015"
```

Př. 6.1.9 Vysvětlete význam a najděte příklady použití následujících funkcí:

a) `show :: Show a => a -> String`
b) `read :: Read a => String -> a`
c) `fromIntegral :: (Integral a, Num b) => a -> b`
d) `round :: (RealFrac a, Integral b) => a -> b`

6.2 Opakování základních funkcí

Př. 6.2.1 Bez použití interpretu určete, jak se vyhodnotí následující výrazy, a určete rovněž jejich nejobecnější typ (bez typových tříd, které jsou implikovány jinými typovými třídami).

a) `last [42, 3.14, 16]`
b) `zipWith mod [3, 5, 7, 4] [4, 2, 3]`
c) `(concat . map (replicate 4)) ['a', 'b', 'c']`
d) `head (filter ((> 3) . length) ["hi!", "ahoj", "hello"])`
e) `cycle [3, 2, 4] !! 10`
f) `(head . drop 3 . iterate (^ 2)) 2`

Př. 6.2.2 Uvažte následující datový typ definující zarovnání textu:

```
data PadMode = PadLeft
             | PadRight
             | PadCenter
             deriving Show
```

Definujte funkci `pad :: Int -> Char -> PadMode -> String -> String`, která na základě zadané specifikace zarovná řetězec zleva, zprava, nebo na střed (v případě, že text nelze vycentrovat přesně, udělejte levé doplnění delší). První argument značí počet znaků, na které se má zarovnat, druhý pak znak, kterým se doplňují kratší řetězce, a třetí směr zarovnávání. Pokud je řetězec delší, než zadaná délka zarovnání, vrátí jej funkce celý. V maximální míře používejte standardní funkce. Existuje řešení bez použití explicitní rekurze.

```
pad 7 '-' PadLeft   "ahoj" ⇝* "---ahoj"
pad 7 '-' PadRight  "ahoj" ⇝* "ahoj---"
pad 7 '-' PadCenter "ahoj" ⇝* "--ahoj-"
pad 3 '-' PadLeft   "ahoj" ⇝* "ahoj"
```

Př. 6.2.3 Redefinujte standardní funkci `take` za pomoci rekurze a bez použití jiných funkcí. Funkci si nejprve celou napište bez použití interpretu, následně si ji v interpretu otestuje a porovnejte chování vaší funkce s tou standardní. *Poznámka*: aby nedocházelo ke konfliktu

jmen, pojmenujte si svou implementaci například `take'`.

Př. 6.2.4 6-breaks 

Naprogramujte funkci `breaks :: (a -> Bool) -> [a] -> [[a]]`, která rozdělí seznam podle zadaného predikátu a prvky odpovídající tomuto predikátu z výsledku vynechá. Při řešení využijte knihovní funkci `break` (dokumentace).

```
breaks (== ' ') "ahoj svete" ⇝* ["ahoj", "svete"]
breaks (== ',') "a,b,,,cde,fg" ⇝* ["a", "b", "", "", "cde", "fg"]
breaks even [1..10] ⇝* [[1], [3], [5], [7], [9]]
breaks even [1, 3, 5, 2, 7] ⇝* [[1, 3, 5], [7]]
breaks even [1, 2, 2, 5] ⇝* [[1], [], [5]]
```

6.3 Vlastní datové typy

Př. 6.3.1

Uvažte následující (parametrizovaný) datový typ stromů s ohodnocenými listy:

```
data BinLeafTree a = LLeaf a
                   | LNode (BinLeafTree a) (BinLeafTree a)
                   deriving (Show, Eq)
```

a) Uveďte alespoň dva různé příklady hodnot tohoto typu a jejich konkrétní typy.

b) Napište všechny typové a všechny hodnotové konstruktory definované touto definicí a určete jejich arity.

c) Naprogramujte funkci `ltSumEven`, která sečte všechna sudá čísla ze všech listů celočíselných stromů. Určete i její typ tak, aby byl nejobecnější možný.

Př. 6.3.2 

```
data RoseLeafTree a = RLNode [RoseLeafTree a]
                    | RLLeaf a
                    deriving (Show, Eq)
```

a) Napište všechny typové a všechny hodnotové konstruktory definované touto definicí a určete jejich arity.

b) Naprogramujte funkci `countValueLeaves`, která spočítá počet listů obsahujících hodnotu (`RLLeaf`). Určete i její nejobecnější možný typ.

c) Naprogramujte funkci `rlFilter`, která je obdobou funkce `filter` pro tyto stromy s tím, že pokud nějaký vrchol nemá po filtraci pod sebou žádné hodnoty, bude rovněž vynechán (pokud se nejedná o kořen). Určete i její nejobecnější možný typ.

```
countValueLeaves (RLLeaf 3) ⇝* 1
countValueLeaves (RLNode [RLLeaf 4, RLLeaf 3, RLLeaf 2]) ⇝* 3
countValueLeaves (RLNode [RLNode [RLNode []], RLLeaf 42]) ⇝* 1

rlFilter even (RLLeaf 3) ⇝* RLNode []
rlFilter even (RLNode [RLLeaf 4, RLLeaf 3, RLLeaf 2])
    ⇝* RLNode [RLLeaf 4, RLLeaf 2]
rlFilter even (RLNode [RLLeaf 3, RLLeaf 5, RLLeaf 7])
    ⇝* RLNode []
rlFilter even (RLNode [RLLeaf 3, RLNode [RLLeaf 5, RLLeaf 7], RLLeaf 1])
    ⇝* RLNode []
rlFilter even (RLNode [RLLeaf 3, RLNode [RLLeaf 5, RLLeaf 7], RLLeaf 2])
```

```
    ⇝* RLNode [RLLeaf 2]
rlFilter (< 4) (RLNode [RLLeaf 3, RLNode [RLLeaf 5, RLLeaf 7], RLLeaf 2])
    ⇝* RLNode [RLLeaf 3, RLLeaf 2]
rlFilter (const True) (RLNode [RLNode [RLNode []], RLLeaf 42])
    ⇝* RLNode [RLLeaf 42]
```

6.4 Typové třídy podrobněji

Př. 6.4.1 S pomocí příkazu `:i` interpretu GHCi nebo dokumentace určete vztahy mezi typovými třídami `Num`, `Integral`, `Eq`, `Ord`, `Show`.

Př. 6.4.2 Uvažte datový typ představující slovníkovou položku zadefinovaný níže.

```
data Entry = Word String
```

Umožněte zobrazování hodnot tohoto typu a jejich porovnávání na rovnost. Dvě hodnoty jsou si rovny, pokud jsou jejich řetězce identické *bez ohledu na velikost písmen*. Formát výpisu zvolte sami.

Jinými slovy, napište instanci `Entry` pro typové třídy `Show` a `Eq`.

> Nebojte se využít funkci `toLower` (nebo `toUpper`) z modulu `Data.Char`.

> **Pan Fešák doporučuje:** Abyste zjistili, které funkce je potřeba implementovat, aby se typ stal instancí typové třídy, použijte `:i` pro danou typovou třídu.
>
> Například na základě dotazu `:i Eq` výpis začíná:
>
> ```
> class Eq a where
>     (==) :: a -> a -> Bool
>     (/=) :: a -> a -> Bool
>     {-# MINIMAL (==) | (/=) #-}
> ```
>
> Instanciací typu do typové třídy `Eq` tedy získáte funkce `(==)` a `(/=)`. Zároveň pro instanci je potřeba definovat buď funkci `(==)`, nebo `(/=)`.

Př. 6.4.3 Mějme datový typ `Shape` definovaný následovně:

```
data Shape = Circle Double
           | Rectangle Double Double
           | Point
           deriving Show
```

Naprogramujte následující funkce:

- `isEqual :: Shape -> Shape -> Bool`, která vrátí `True`, právě tehdy, když jsou si oba argumenty rovny.
- `isGreater :: Shape -> Shape -> Bool`, která vrátí `True`, pokud je první argument větší než druhý (`Shape` je větší než druhý, když má větší obsah);

Př. 6.4.4 Uvažte datový typ představující semafor zadefinovaný níže.

```
data TrafficLight = Red | Orange | Green
```

Umožněte zobrazování hodnot tohoto typu, jejich vzájemné porovnávání a řazení (zelená < oranžová < červená). Řečeno jinak, napište instanci `TrafficLight` pro typové třídy

`Show`, `Eq` a `Ord`.

> **Jindřiška varuje:** Nezapomeňte, že funkce, které máte implementovat při psaní instance typu pro danou typovou třídu, si můžete zobrazit pomocí `:i` *`typová třída`*.

Př. 6.4.5 
Zadefinujme vlastní typ uspořádaných dvojic s názvem `PairT`. Tento typ bude mít pouze jeden binární hodnotový konstruktor `PairD` (viz definice níže).

```
data PairT a b = PairD a b
```

Vytvořte instanci `PairT` pro typové třídy `Show`, `Eq` a `Ord`. Ať jsou si dvě dvojice rovny právě tehdy, pokud jsou si rovny po složkách. Uspořádání použijte lexikografické. Zobrazování hodnot tohoto typu nechť je slovní (tedy namísto obligátního `(1, 2)` vypište třeba `"pair of 1 and 2"`).

Př. 6.4.6 
Deklarujte typ `data BinTree a = Empty | Node a (BinTree a) (BinTree a)` jako instanci typové třídy `Eq`. Instanci si napište sami (tj. nepoužívejte klauzuli `deriving`).

*
**

Na konci cvičení byste měli zvládnout:

- otypovat výrazy a funkce, a to včetně polymorfních funkcí využívajících typové třídy a včetně funkcí s více definičními rovnostmi a s vzory;
- z typu poznat, kolik argumentů funkce má;
- z typu poznat, co může funkce dělat;
- umět psát rekurzivní funkce na seznamech a vlastních datových typech, včetně jednodušších funkcí vyšších řádů;
- umět ověřit vztahy mezi typovými třídami;
- a vše ostatní, co bylo v předchozích cvičeních.

Řešení

Řeš. 6.η.1

a) `Bool`

b) `[a]`

c) `[String]` (případně ekvivalentně `[[Char]]`)

Z podvýrazu `"" :: String` víme, že v seznamu musí být řetězce.

d) Nelze otypovat, seznamy musí být homogenní (všechny hodnoty musí mít stejný typ, ale první je typu `Bool` a druhý `[a]`, což jsou neunifikovatelné typy).

e) `Fractional a => [a]`; tedy `a` může být libovolný typ schopný reprezentovat zlomky (popřípadě také `(Num a, Fractional a) => [a]`, ale `Fractional` implikuje `Num`).

Pozor, stále se jedná o seznam, jehož všechny prvky mají stejný typ. Jen dosud není řečeno, jaký to bude, jen že to musí být nějaký typ z třídy `Fractional`. Celá čísla lze samozřejmě reprezentovat i pomocí typů z třídy `Fractional`, např. `Double`.

Jednotlivé prvky seznamu mají typ `1 :: Num a => a`, `2 :: Num a => a` a `3.14 :: Fractional a => a`. Jelikož se mají společně nacházet v jednom seznamu, musíme je unifikovat. Při tom dostaneme kontext `(Num a, Fractional a) =>`, ten však můžeme zjednodušit na `Fractional a =>`, protože `Fractional` je podtřídou `Num`.

f) Nelze otypovat. Chybová hláška `No instance for (Num Bool) arising from the literal '1'` říká, že `Bool` není číslo (instance `Num`). Přesněji vzato, nelze otypovat s definicemi, které máme v `Prelude` (a je celkem dobrý důvod na to, aby tato instance neexistovala).

Typy jednotlivých prvků seznamu jsou `1 :: Num a => a`, `2 :: Num a => a` a `True :: Bool`. Jelikož však `Bool` není číslo, nejde tyto typy unifikovat, a tedy nemohou být společně v seznamu.

g) `(Ord a, Num a) => [a]` (obě části kontextu jsou nutné, mezi `Ord` a `Num` není žádný vztah implikace)

Podvýrazy mají typy `filter :: (a -> Bool) -> [a] -> [a]`, `(\x -> x > 5) :: (Ord n, Num n) => n -> Bool` a `[1, 2, 4, 8] :: Num m => [m]`. Dosadíme-li typy argumentů za odpovídající typy `filter` a provedeme unifikaci, odstraněním již použitých argumentů dostáváme typ `(Ord a, Num a) => [a]`.

h) `(Show a, Integral a) => a -> String`; `Show` je typová třída typů, které lze převést do textové reprezentace.

Popřípadě také `(Show a, Num a, Integral a) => a -> String`, ale `Integral` implikuje `Num`.

i) `(Num a, Read a) => String -> a`; `Read` je typová třída typů, jejichž hodnoty lze parsovat z textové reprezentace (typ výsledku funkce `read` se odvodí z kontextu použití).

j) `Integral a => a -> Double`; explicitní otypování je zde použito k vynucení konverze na konkrétní typ. `fromIntegral :: (Integral a, Num b) => a -> b` totiž umožňuje konverzi celého čísla na libovolný číselný typ.

Řeš. 6.η.2

a) Ze vzoru vidíme, že vstupem je dvojice, a z výrazu na pravé straně vidíme, že i návratový typ je dvojice. Zároveň typ první složky v argumentu musí být stejný jako typ druhé

složky v návratovém typu a naopak. Celkově tedy `swap :: (a, b) -> (b, a)`.

b) První argument musí být typu `Bool` podle prvního řádku definice. Podle prvního řádku tedy dostaneme typ `Bool -> (a, b) -> (b, a)`, zatímco podle druhého `Bool -> (c, d) -> (c, d)`. Jelikož typy na odpovídajících pozicích musíme unifikovat (tedy `(a, b)` ∼ `(c, d)` a `(b, a)` ∼ `(c, d)`), jediná možnost je, že oba typy ve dvojici budou stejné. `swapIf :: Bool -> (a, a) -> (a, a)`.

c) Z toho, že v obou vzorech je právě jeden argument a funkce vrací `String`, vidíme, že nejobecnější možný typ funkce je `a -> String`. Typ argumentů funkce však není závislý jen na jejich použití na pravé straně definice, ale i na vzorech. Jelikož `[]` je vzor prázdného seznamu, musí být argument funkce seznamového typu. Další omezení již nejsou, dostáváme tedy `sayLength :: [a] -> String`.

d) Z použitých vzorů můžeme odvodit, že funkce bere dva argumenty a že první je typu `Char`. Z návratové hodnoty prvního řádku můžeme odvodit typ `Bool`. Zbývá už jen určení typu druhého argumentu. Ve druhém vzoru si můžeme všimnout, že vracíme hodnotu, kterou bereme ve druhém argumentu. Takže obě mají stejný typ, a protože návratová hodnota má typ `Bool`, i druhý argument bude mít typ `Bool`. Dostáváme tedy `aOrX :: Char -> Bool -> Bool`.

Řeš. 6.η.3 Máme dvě možnosti funkcí, které svůj argument ignorují, a dvě varianty pro funkce, které s argumentem nějak pracují – identitu a negaci.

```
bfalse :: Bool -> Bool
bfalse _ = False

btrue :: Bool -> Bool
btrue _ = True

bid :: Bool -> Bool
bid x = x

bnot :: Bool -> Bool
bnot True  = False
bnot False = True
```

Řeš. 6.η.5

```
binmap :: (a -> a -> b) -> [a] -> [b]
binmap f xs = zipWith f xs (tail xs)

binmap' :: (a -> a -> b) -> [a] -> [b]
binmap' _ []       = []
binmap' _ [_]      = []
binmap' f (x:y:xs) = f x y : binmap f (y:xs)
```

Řeš. 6.1.1

a) Arita 3. Po aplikaci na tři argumenty libovolného, ale stejného typu, který navíc musí být instancí typové třídy `Eq`, je výsledek typu `Bool`. Příkladem funkce, která může mít tento typ, je `\x y z -> x == y && y == z`.

b) Arita 2. Závorka vpravo (kolem `[a] -> Int`) nemá žádný efekt – tato závorka by v typu implicitně byla, i kdybychom ji vynechali a odpovídá částečné aplikaci. Prvním argumentem je funkce `a -> Bool`, druhým je pak seznam, jehož prvky mají stejný typ jako argumenty funkce v prvním argumentu. Výsledek je pak `Int`. Například `(\p -> length . filter p) :: (a -> Bool) -> ([a] -> Int)`.

c) Arita 0. Nejedná se o funkci (je to seznam funkcí). Například `[(+), (-)] :: [Int -> Int -> Int]`.

d) Arita 3. Po aplikaci na 3 argumenty typů `Int`, `Integer` a `String` dostaneme výsledek typu `String`. Například `(\x y z -> show x ++ " " ++ show y ++ " " ++ z) :: Int -> Integer -> String -> String`.

e) Arita 2. První argument je funkce typu `Int -> Integer`, druhý je `String`, výsledek je `String`. Například `(\f x -> show (f (read x))) :: (Int -> Integer) -> String -> String`.

f) Arita 3. Tento typ je ekvivalentní typu v d) (uzávorkování funkčního typu zprava odpovídá částečné aplikaci funkce).

g) Arita 2. Bere dva argumenty (`Int` a `Integer`) a produkuje seznam funkcí (typu `[String -> String]`). Například `(\x y -> [ \s -> show x ++ show i ++ s | i <- [1..y] ]) :: Int -> Integer -> [String -> String]`.

h) Aritu nelze z typu určit, bude ale nejméně 3. Důvodem je, že na místě výsledku je v typu typová proměnná `c`, za kterou ale můžeme dosadit i funkční typ. Víme však, že funkce bude brát minimálně tři argumenty – funkci typu `a -> b -> c` a následně dva argumenty typů `b` a `a` (které opět mohou být i funkce).

Uvažme příklady konkrétních typů, které můžeme za jednotlivé argumenty dosadit.

- Za první argument dosadíme typ funkce `(||) :: Bool -> Bool -> Bool` – druhý i třetí argument pak musí být `Bool` a stejně tak i výsledek. Pro tuto konkretizaci dostáváme tedy aritu 3.
- Za první argument dosadíme typ funkce `zipWith :: (d -> e -> f) -> [d] -> [e] -> [f]`. Ten můžeme uzávorkovat jako `zipWith :: (d -> e -> f) -> [d] -> ([e] -> [f])` a položit `a` ≡ `d -> e -> f`, `b` ≡ `[d]` a `c` ≡ `[e] -> [f]`. Tím dostaneme konkretizaci našeho typu na `((d -> e -> f) -> [d] -> [e] -> [f]) -> [d] -> ((d -> e -> f) -> [d] -> [e] -> [f]) -> [e] -> [f]`, a tedy aritu 4.

Konkrétním (a jediným totálním) příkladem funkce s tímto typem je `flip`. Můžete si v interpretru ověřit, že počet argumentů této funkce nutných k tomu, aby výsledek nebyla funkce, je vskutku závislý na prvním argumentu. Uvážit můžeme třeba `flip (||) True False :: Bool` a `flip zipWith [1, 2, 3] (+) [3, 2, 1] :: Num a => [a]`.

Řeš. 6.1.2 a) `cm :: (a -> [b]) -> [a] -> [b]`
Při typování rekurzivních funkcí může být výhodné dívat se nejprve na bázový příklad. V tomto případě z něj však moc nezjistíme: vidíme, že má typ `a -> [b] -> [c]`, tedy víme jen, že druhý argument je seznam a návratová hodnota je taktéž seznam. Z rekurzivní části definice pak plyne, že typ návratové hodnoty celé funkce musí být stejný jako typ návratové hodnoty funkce `f` (plyne z typu ++), a že funkce `f` musí brát jako argumenty hodnoty ze seznamu v druhém argumentu `cm`.
Funkce je podobná funkci `map`, ale z každého prvku původního seznamu vytvoří seznam prvků a tyto seznamy spojí. Najdeme ji i mezi základními funkcemi v Haskellu pod názvem `concatMap`.

b) `mm :: (Bounded a, Ord a) => [a] -> (a, a)`
Z bázového případu (prvního řádku) odvodíme, že výsledek bude dvojice. Dále z typů

`minBound :: Bounded a => a` a `maxBound :: Bounded a => a` odvodíme, že obě složky výsledné dvojice budou instancemi typové třídy `Bounded` (ale zatím nic neříká, že to musí být stejné typy). Ze vzoru odvodíme, že argument bude seznam. Dohromady tedy z bázového případu dostáváme typ `(Bounded b, Bounded c) => [a] -> (b, c)`.
Z rekurzivního případu pak opět vidíme, že argument je seznam a výsledek dvojice – to je konzistentní s již zjištěným typem. Dále ale vidíme, že obě složky výsledné dvojice musí být stejného typu jako prvky vstupu a navíc tento typ musí být instancí `Ord` – obojí plyne z typu `max :: Ord a => a -> a -> a`, resp. `min` stejného typu. Z druhého řádku definice tedy plyne typ `Ord a => [a] -> (a, a)`.
Typy získané z jednotlivých řádků definice unifikujeme a při tom i sloučíme kontexty. Tím dostaneme výsledný typ.
Funkce `mm` počítá minimum a maximum z hodnot v seznamu a dělá to v jednom průchodu. Navíc funguje i na prázdném seznamu, kde jako minimum vrací nejvyšší hodnotu daného typu a jako maximum nejnižší – tyto hodnoty můžeme chápat jako neutrální prvky pro minimum (resp. maximum) pokud je typ omezený. Tato funkce tedy nefunguje s neomezenými typy jako je `Integer`.

c) `c :: (b -> c) -> (a -> b) -> a -> c`
Funkci můžeme ekvivalentně zapsat jako

```
c' f g x = f (g x)
```

Jedná se tedy o funkci, která bere dvě funkce a hodnotu libovolného typu a aplikuje funkce postupně na tuto hodnotu. Tedy o ekvivalent funkce `(.)`.
Můžeme postupovat i více algoritmicky:

1. můžeme začít s tím, že parametrům (včetně toho v lambda funkci) dáme libovolný typ.

   ```
   x :: t
   y :: u
   z :: a
   ```

2. dále si všimneme, že `y` je aplikováno na `z`, a tedy se musí jednat o funkci, která bere argumenty typu `a`; výsledek je zatím neznámého typu

   ```
   x :: t
   y :: a -> b
   z :: a
   ```

3. z toho tedy plyne, že `(y z) :: b`
4. dále pak `x` je aplikováno na `(y z)`, tedy to opět musí být funkce, tentokrát beroucí argument typu `b`; výsledek je opět zatím neznámého typu

   ```
   x :: b -> c
   y :: a -> b
   z :: a
   ```

5. tedy `x (y z) :: c`;
6. dále můžeme otypovat lambda funkci na pravé straně definice funkce ze zadání: `(\z -> x (y z)) :: a -> c`, výsledek je samozřejmě typ výsledku na pravé straně lambda funkce, zatímco před `->` je typ argumentu;
7. nyní již zbývá přidat jen typy argumentů `x` a `y`: `c :: (b -> c) -> (a -> b) -> a -> c`.

Řeš. 6.1.4

a) Arita 2. Funkce `map` bere jako první argument funkci, druhý pak seznam (jehož prvky mají stejný typ jako vstup funkce). Funkce `flip` pak pouze prohodí pořadí argumentů, takže seznam bude první a funkce druhá. Produkuje seznam.

b) Arita 1. Vstupem je seznam, výstup je `Bool`.

c) Arita 3. Kromě dvou argumentů explicitně zapsaných lambda funkcí (typů `a -> b`, a `b -> Bool`) bere tato funkce ještě seznam (`[a]`).

d) Arita 1. Nejobecnější typ výrazu je `(\x -> x 1 || x 2) :: Num a => (a -> Bool) -> Bool`. Jediným vstupem je tedy funkce (predikát na číslech).

e) Arita 2. Argument `flip` je operátorovou sekcí od `(.)`, druhým argumentem bude funkce, prvním pak hodnota, která může vstoupit do této funkce. `flip (not .) :: a -> (a -> Bool) -> Bool`.

f) Arita 2. Oba argumenty jsou seznamy (to plyne již z typu `zipWith`), výsledkem je rovněž seznam. Jediná potíž by mohla být, pokud by tento výraz nešel otypovat. Zde je však třeba si uvědomit, že `id` je polymorfní funkce typu `a -> a` a za `a` lze tedy dosadit i funkční typ (třeba `b -> c`, a tak dostat specializaci na `(b -> c) -> b -> c`). Dostáváme tedy `zipWith id :: [b -> c] -> [b] -> [c]`, tedy funkci, která vezme seznam funkcí a seznam argumentů, a aplikuje funkce na argumenty.

Řeš. 6.1.5 Naše funkce bere dva argumenty – funkci (pojmenujeme `f`) a hodnotu (pojmenujeme `x`). Z typu funkce víme, `x` lze dát jako argument `f`. Musíme si rozmyslet co s nimi můžeme dělat pro to, abychom vrátili něco typu `b`.

Dále se můžeme zamyslet nad tím, co můžeme dělat s argumentem typu `a`. Jelikož však tento argument může být *libovolného* typu, nevíme nic o jeho vlastnostech a struktuře a nemáme ho tedy jak modifikovat.

Zároveň potřebujeme nějak vyrobit hodnotu typu `b`. Opět o tomto typu nic nevíme, a tedy nemáme jinou možnost jak ho vyrobit, než pomocí `f`.

Existuje tedy jen jedna totální funkce tohoto typu:

```
app :: (a -> b) -> a -> b
app f a = f a
```

Tato funkce je ekvivalentní standardnímu operátoru `($)`.

Řeš. 6.1.6 Žádná totální funkce tohoto typu neexistuje. Je třeba si uvědomit, že aby taková totální funkce existovala, musela by být schopná vytvořit hodnotu typu `a` z prázdného seznamu. To však není možné, máme-li neznámý typ `a` nemáme jak vytvořit jeho hodnotu.

Řeš. 6.1.7 Uvažme funkci `foo :: (a -> Maybe b) -> Maybe a -> Maybe b`. Tu pak můžeme volat jako `foo f x`, kde `f` je funkce typu `a -> Maybe b` a `x` je hodnota typu `Maybe a`. Uvažme jaká chování může funkce `foo` mít.

a) Nezávisle na argumentech může vrátit `Nothing`.

```
constNothing :: (a -> Maybe b) -> Maybe a -> Maybe b
constNothing _ _ = Nothing
```

b) Pokud chceme vyprodukovat nějakou jinou hodnotu než `Nothing`, máme k dispozici pouze dané argumenty `f` a `x`. Speciálně nemáme jak vytvořit hodnotu typu `b` jinak, než pomocí funkce `f :: a -> Maybe b` – důvodem je to, že `b` může být libovolný typ a nic o něm nevíme.

Pro použití funkce `f` potřebujeme hodnotu typu `a`. Tu můžeme získat jedině z hodnoty `x :: Maybe a`. Ačkoli o `a` nevíme nic, o hodnotě typu `Maybe a` již víme, že může být `Nothing` nebo `Just z` pro nějaké `z`.

Můžeme tedy použít vzory pro rozložení hodnoty `x` a pokud není `Nothing`, použít uvnitř uloženou hodnotu typu `a` jako vstup do funkce `f`.

```
bind :: (a -> Maybe b) -> Maybe a -> Maybe b
bind _ Nothing  = Nothing
bind f (Just x) = f x
```

c) Žádná další možnost není.

Řeš. 6.1.8 Prvně otypujeme řádky jednotlivě:

1. První argument musí jistě být číslo a porovnatelný, typ druhého argumentu musí být seznamový, protože se objevuje v `then` větvi `if`u, kde se v `else` větvi objevuje seznam, typ třetího argumentu je `Bool`. Návratová hodnota je seznam stejného typu jako druhý argument, protože můžeme vracet přímo druhý argument.

   Dostáváme tedy `(Num a, Ord a) => a -> [b] -> Bool -> [b]`

2. Z druhého řádku o prvním argumentu nevíme nic, o druhém víme, že je to seznam a že je stejného typu jako návratová hodnota. O třetím argumentu opět víme, že je to `Bool`

   `c -> [d] -> Bool -> [d]`

3. O argumentech nevíme nic, ale víme, že návratová hodnota je řetězec.

   `e -> f -> g -> String`

V těchto případech je vhodné nechat zatím typové proměnné v jednotlivých typech různé, abychom zabránili náhodnému propojení typů, které spolu nesouvisí.

Nyní zbývá unifikovat typy na pozicích, které si v definici funkce odpovídají.

- `a` ∼ `c` ∼ `e`, zde nesmíme zapomenout, že s `a` se pojí typový kontext. Unifikace je naopak jednoduchá, protože unifikuje jen samotné typové proměnné. Nadále budeme místo nich všech používat `a` (substituce `c` ↦ `a` a `e` ↦ `a`).
- `[b]` ∼ `[d]` ∼ `f` vyřešíme substitucí `d` ↦ `b` a `f` ↦ `[b]`.
- `Bool` ∼ `Bool` ∼ `g`, zde je substituce jednoduchá: `g` ↦ `Bool`.
- `[b]` ∼ `[d]` ∼ `String`, což už ale máme substituováno za `[b]` ∼ `[b]` ∼ `String` (protože `d` se nahradilo za `b`).

  Toto na první pohled nevypadá moc dobře, protože se zdánlivě snažíme unifikovat seznam s něčím, co není seznam. Avšak `String` je jen alias pro `[Char]`, a tedy unifikovat se seznamovými typy jej lze. Dostáváme `b` ↦ `Char`.

Nyní je třeba provést substituce v typech z jednotlivých řádků definice. Nemělo by záležet na tom, který řádek vezmeme, za předpokladu, že substituujeme, dokud můžeme. Celkově dostaneme `f :: (Num a, Ord a) => a -> [Char] -> Bool -> [Char]`, neboli `f :: (Num a, Ord a) => a -> String -> Bool -> String`. Je důležité nezapomenout na kontext svázaný s typovou proměnnou `a`.

Řeš. 6.1.9

a) Jedná se o funkci, která dokáže (pro typy, které to umožňují) převést hodnoty daného typu na jejich textovou reprezentaci. Např. `show 42` $\rightsquigarrow^*$ `"42"`, `show [16, 42]` $\rightsquigarrow^*$ `"[16,42]"`. Funguje pro většinu typů s výjimkou funkčních typů. Textová forma vyprodukované `show` by typicky měla být zápis validního Haskellového výrazu.

b) Jedná se o funkci, která převádí textovou reprezentaci na hodnotu požadovaného typu. Typ výsledné hodnoty se odvodí z použití funkce `read`, v případě potřeby je možné jej vynutit explicitním otypováním:

```
(read "42" :: Int) ⇝* 42
(read "42" :: Float) ⇝* 42.0
(read "[1, 2, 3, 4]" :: [Int]) ⇝* [1, 2, 3, 4]
read "40" + read "2" ⇝* 42
```

c) Funkce pro převod celého čísla na libovolnou reprezentaci čísla, např. funkci pro výpočet e-té odmocniny čísla n, kde e je celé číslo a n je číslo s plovoucí desetinnou čárkou (např. `Double`), lze zapsat jako:

```
root :: (Floating a, Integral b) => a -> b -> a
root n e = n ** (1 / fromIntegral e)
```

(Operátor `(**)` slouží k umocňování čísel s plovoucí desetinnou čárkou.)

d) Funkce pro zaokrouhlování desetinných čísel na celá čísla (existuje i `floor` a `ceiling`). Např. `round 1.6` $\rightsquigarrow^*$ `2`, `floor 1.6` $\rightsquigarrow^*$ `1`. (Typová třída `RealFrac` obsahuje právě čísla, která lze zaokrouhlovat, z běžných typů do ní patří `Float` a `Double`.)

Řeš. 6.2.1

a) `last [42, 3.14, 16]` $\rightsquigarrow^*$ `16.0 :: Fractional a => a`

b)
```
zipWith mod [3, 5, 7, 4] [4, 2, 3]
      ⇝* [3, 1, 1] :: Integral a => [a]
```

c)
```
(concat . map (replicate 4)) ['a', 'b', 'c']
      ⇝* concat (map (replicate 4) ['a', 'b', 'c'])
      ⇝* "aaaabbbbcccc" :: String
```

d)
```
head (filter ((> 3) . length) ["hi!", "ahoj", "hello"])
      ⇝* head ["ahoj", "hello"]
      ⇝* "ahoj" :: String
```

e) `cycle [3, 2, 4] !! 10` $\rightsquigarrow^*$ `2 :: Num a => a`

f)
```
(head . drop 3 . iterate (^ 2)) 2
      ⇝* ((2 ^ 2) ^ 2) ^ 2
      ⇝* 256 :: Num a => a
```

Řeš. 6.2.2

```
pad :: Int -> Char -> PadMode -> String -> String
pad padTo padChar padMode str = leftPad padMode
                              ++ str
                              ++ rightPad padMode
  where
    diff = max 0 (padTo - length str)
    rdiff = diff `div` 2
    ldiff = diff - rdiff
```

```
mkPad n = replicate n padChar

leftPad PadLeft   = mkPad diff
leftPad PadCenter = mkPad ldiff
leftPad PadRight  = ""

rightPad PadLeft   = ""
rightPad PadCenter = mkPad rdiff
rightPad PadRight  = mkPad diff
```

Řeš. 6.2.3

```
take' :: Int -> [a] -> [a]
take' _ [] = []
take' 0 _  = []
take' amount (x:xs) = x : (take' (amount - 1) xs)
```

Řeš. 6.3.1

a) Např.

- `LLeaf 42 :: BinLeafTree Int`
- `LNode (LLeaf 3) (LLeaf 0.2) :: BinLeafTree Double`
- `LNode (LLeaf "hi") (LLeaf []) :: BinLeafTree String`

b) Typový konstruktor `BinLeafTree` arity 1. Hodnotové konstruktory `LLeaf` (arity 1) a `LNode` (arity 2).

c)
```
ltSumEven :: Integral a => BinLeafTree a -> a
ltSumEven (LLeaf x)   = if even x then x else 0
ltSumEven (LNode l r) = ltSumEven l + ltSumEven r
```

Řeš. 6.3.2

a) Typové: `RoseLeafTree` arity 1. Hodnotové: `RLNode` a `RLLeaf`, oba arity 1.

b)
```
countValueLeaves :: Integral i => RoseLeafTree a -> i
countValueLeaves (RLLeaf _)  = 1
countValueLeaves (RLNode ls) = sum (map countValueLeaves ls)
```

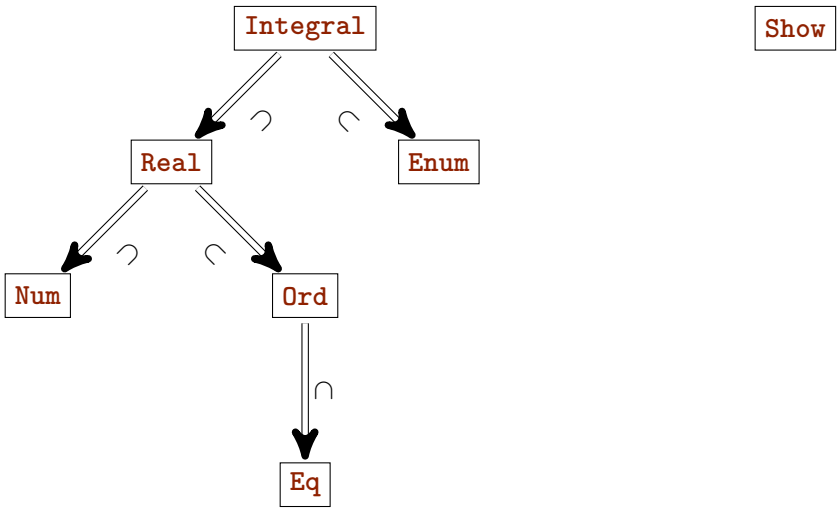
Čistě z hlediska Haskellu by nejobecnější možný typ této funkce měl kontext `Num a =>` místo `Integral a =>`, ale vzhledem k tomu, že výsledek má být počet, je vhodné, aby byl jeho typ celočíselný.

c) Je dobré si všimnout, že podmínka na prořezání stromu není závislá na tom, zda byl list odstraněn díky filtraci, nebo vůbec nebyl ve vstupu. Zároveň podle prvního příkladu se samotný list nahrazuje za uzel bez následníků. Těchto dvou vlastností můžeme využít a nejprve převést listy, které nechceme, na uzly bez následníků a následně tyto uzly prořezat. Ke zpracování seznamů v uzlech použijeme `map` a `filter`.

```
rlFilter :: (a -> Bool) -> RoseLeafTree a -> RoseLeafTree a
rlFilter p (RLLeaf x)  = if p x then RLLeaf x else RLNode []
rlFilter p (RLNode ls) =
    RLNode (filter nontrivial (map (rlFilter p) ls))
  where
    nontrivial (RLNode []) = False
    nontrivial _           = True
```

Řeš. 6.4.1 Vztahy lze vyčíst z hlavičky třídy zobrazené příkazem `:i`, např. `class (Real a, Enum a) => Integral a where` říká, že třídy `Read` a `Enum` jsou nadtřídami `Integral`. Jinak řečeno, každý typ, který je v `Integral`, musí být i v `Real` a `Enum`. Na třídy se také můžeme dívat jako na množiny typů – `Integral` je podmnožinou `Real`.

Kompletní vztahy mezi těmito třídami lze znázornit následujícím diagramem:



Je dobré si povšimnout následujícího:

- žádná z číselných tříd automaticky nevyžaduje zobrazitelnost pomocí `show`;
- obecně čísla (jiná než reálná) nemusí jít porovnávat/řadit – například komplexní čísla nemohou být instancí `Ord`; u některých reprezentací čísel (např. spočetná čísla[a] nebo čísla zadána pomocí výrazů, rovnic či funkcí) nemusí dávat dobrý smysl ani rovnost;
- všechny celočíselné typy jsou i instancemi `Enum`, což mimo jiné umožňuje používat zápisy jako `[1..42]`; dalším příkladem typu, který je instancí `Enum`, je `Bool`.

[a] Anglicky computable numbers, jsou čísla pro něž můžeme algoritmicky spočítat desetinný rozvoj do libovolné přesnosti.

Řeš. 6.4.2

```
import Data.Char

instance Show Entry where
    show (Word str) = show str

instance Eq Entry where
    (Word str1) == (Word str2) = lowStr str1 == lowStr str2
      where
        lowStr xs = map toLower xs
```

Řeš. 6.4.3

```
isEqual :: Shape -> Shape -> Bool
isEqual (Circle r1)       (Circle r2)       = r1 == r2
isEqual (Rectangle a1 b1) (Rectangle a2 b2) = a1 == a2 && b1 == b2
isEqual Point             Point             = True
isEqual _                 _                 = False

isGreater :: Shape -> Shape -> Bool
isGreater shape1 shape2 = area shape1 > area shape2
    where
        area (Circle r)      = pi * r * r
```

```
    area (Rectangle a b) = a * b
    area Point           = 0
```

Řeš. 6.4.4

```
instance Eq TrafficLight where
    Red == Red       = True
    Orange == Orange = True
    Green == Green   = True
    _ == _           = False

instance Ord TrafficLight where
    Green <= _       = True
    _ <= Red         = True
    Orange <= Orange = True
    _ <= _           = False

instance Show TrafficLight where
    show Red    = "červená"
    show Orange = "oranžová"
    show Green  = "zelená"
```

Řeš. 6.4.5

```
instance (Eq a, Eq b) => Eq (PairT a b) where
    PairD a1 b1 == PairD a2 b2 = a1 == a2 && b1 == b2

instance (Ord a, Ord b) => Ord (PairT a b) where
    PairD a1 b1 <= PairD a2 b2 = a1 < a2 || (a1 == a2 && b1 <= b2)

instance (Show a, Show b) => Show (PairT a b) where
    show (PairD a b) = "pair of " ++ show a ++ " and " ++ show b
```

Řeš. 6.4.6

```
instance Eq a => Eq (BinTree a) where
    Empty          == Empty          = True
    Node x1 l1 r1 == Node x2 l2 r2 =
        x1 == x2 && l1 == l2 && r1 == r2
    _              == _              = False
```

Poslední řádek nelze vynechat – pokrývá porovnávání prázdného a neprázdného stromu.

Přiložený kód

> **Pan Sazeč upozorňuje:** Kopírování kódu ze souboru PDF nezachovává odsazení! Soubory jsou ale vloženy i jako přílohy tohoto dokumentu; s rozumným prohlížečem je můžete stáhnout kliknutím na název souboru, ať už zde, či v příslušných příkladech. Přílohy jsou k nalezení také ve studijních materiálech v ISu.

📎 06_data.hs

```haskell
-- typy a typové třídy: na papír, viz sbírka

-- 6.1.7: tento typ si zkopírujte na všechny funkce podle příkladu
fMaybe :: (a -> Maybe b) -> Maybe a -> Maybe b
fMaybe = undefined

data BinLeafTree a = LLeaf a
                   | LNode (BinLeafTree a) (BinLeafTree a)
                   deriving (Show, Eq)

-- je třeba doplnit i typ:
-- bitLeafTreeVal1
-- bitLeafTreeVal2

-- typové konstruktory:
-- hodnotové konstruktory:

-- je třeba doplnit i typ:
-- tSumEven

data RoseLeafTree a = RLNode [RoseLeafTree a]
                    | RLLeaf a
                    deriving (Show, Eq)

-- typové konstruktory:
-- hodnotové konstruktory:

-- je třeba doplnit i typ:
-- countValueLeaves

-- je třeba doplnit i typ:
-- rlFilter
```