Cvičení 5: Intensionální seznamy, lenost, foldy

Před pátým cvičením je zapotřebí znát:

- zápisy hromadným výčtem jako `[1..5]`, `[1,3..100]`, `[0,-2..10]`;
- co jsou intensionální seznamy a jak se v Haskellu zapisují (kvalifikátory, generátory, predikáty, lokální definice), tj. například umět přečíst zápis `[ 2 * y | x <- [1,2,3,4,5], even x, let y = 2 + x ]`;
- co je to vyhodnocovací strategie;
- jak probíhá striktní a líné vyhodnocování;
- jak fungují akumulační funkce na seznamech, tj. funkce:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr1 :: (a -> a -> a) -> [a] -> a
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl1 :: (a -> a -> a) -> [a] -> a
```

Etudy

Etuda 5.η.1 Pomocí intensionálního seznamu definujte funkci `divisors :: Integer -> [Integer]`, která k zadanému kladnému celému číslu vrátí seznam jeho kladných dělitelů (bez duplikátů).

```
divisors 1  ⇝* [1]
divisors 4  ⇝* [1, 2, 4]
divisors 12 ⇝* [1, 2, 3, 4, 6, 12]
```

Etuda 5.η.2 Uvažme bilance kreditních karet osob reprezentované seznamem dvojic (*jméno vlastníka*, *množství peněz*). Pomocí intensionálního seznamu definujte funkci `negativeCredit :: [(String, Integer)] -> [String]`, která pro takový seznam karet vrátí právě ta jména, na jejichž kartě je záporná částka.

```
negativeCredit [("James", 0), ("Mikael", -28), ("Eric", 18),
                ("Heather", 10), ("Irene", 7), ("Magnus", -4),
                ("Devon", -14), ("Devin", -30)]
    ⇝* ["Mikael", "Magnus", "Devon", "Devin"]
```

Poznámka: V řešení skutečně smysluplně použijte intensionální seznam, nepoužívejte explicitní rekurzi ani funkce jako `map` a `filter`.

Etuda 5.η.3 Pomocí intensionálního seznamu definujte funkci `allDivisibleSums`, která ze zadaných dvou seznamů celých čísel `xs` a `ys` a kladného čísla `d` vytvoří seznam všech možných trojic (*číslo z* `xs`, *číslo z* `ys`, *součet prvních dvou složek*) splňujících, že součet je dělitelný číslem `d`. Pro každou dvojici čísel počítejte součet právě jednou.

```
allDivisibleSums [0, 1, 2] [3, 4, 5] 3 ⇝*
    [(0, 3, 3), (1, 5, 6), (2, 4, 6)]
allDivisibleSums [0, 1] [2, 3] 1 ⇝*
    [(0, 2, 2), (0, 3, 3), (1, 2, 3), (1, 3, 4)]
allDivisibleSums [0, 1] [2, 3] 7 ⇝* []
```

Etuda 5.η.4 Definujme si typový alias `UCO` ekvivalentní typu `Int`:

```
type UCO = Int
```

Reprezentujme seznam studentů a jimi absolvovaných předmětů pomocí typu `[(UCO, [String])]`, kde první složka dvojice reprezentuje učo studenta a druhá složka kódy předmětů, které daný student absolvoval.

Pomocí intensionálních seznamů (bez explicitní rekurze a funkcí jako `map` a `filter`) napište funkce:

a) `countPassed :: [(UCO, [String])] -> [(UCO, Int)]`, která vrátí učo každého studenta spolu s počtem předmětů, které daný student absolvoval,

b) `atLeastTwo :: [(UCO, [String])] -> [UCO]`, která vrátí uča studentů, kteří absolvovali alespoň dva předměty,

c) `passedIB015 :: [(UCO, [String])] -> [UCO]`, která vrátí uča studentů, kteří absolvovali předmět `"IB015"` (může se vám hodit funkce `elem`),

d) `passedBySomeone :: [(UCO, [String])] -> [String]`, která vrátí kódy předmětů, které absolvoval alespoň jeden student (kódy se v seznamu můžou opakovat).

> **Pan Fešák doporučuje:** Nezapomeňte, že v intensionálních seznamech lze využívat vzory.

```
students :: [(UCO, [String])]
students = [(415409, []), (448093, ["IB111", "IB015", "IB000"]),
            (405541, ["IB111", "IB000", "IB005", "MB151", "MB152"])]

countPassed students     ⇝* [(415409, 0), (448093, 3), (405541, 5)]
atLeastTwo students      ⇝* [448093, 405541]
passedIB015 students     ⇝* [448093]
passedBySomeone students ⇝* ["IB111", "IB015", "IB000", "IB111", "IB000",
                             "IB005", "MB151", "MB152"]
```

Etuda 5.η.5 Naprogramujte unární funkci `naturalsFrom :: Integer -> [Integer]`, která pro vstup `n` vrátí nekonečný seznam `[n, n + 1, n + 2,...]`.

Pomocí funkce `naturalsFrom` definujte nekonečný seznam `naturals :: [Integer]` všech přirozených čísel včetně nuly.

> **Pan Fešák doporučuje:** Vyhodnocovat nekonečné seznamy celé není úplně praktické. Může se vám hodit klávesová zkratka Ctrl+C, která zabije aktuální výpočet a hlavně funkce `take :: Int -> [a] -> [a]`, například si můžete vyhodnotit `take 3 (naturalsFrom 100)` ⇝* `[100, 101, 102]`.

Etuda 5.η.6 S pomocí rekurze naprogramujte funkci `maxmin :: Ord a => [a] -> (a, a)`, která pro zadaný seznam *v jednom průchodu* zjistí jeho maximum a minimum. Předpokládejte, že vstupní seznam je neprázdný.

Nápověda: může se vám hodit pomocná funkce s tzv. akumulátorem – dodatečným argumentem sloužícím pro postupné vytváření výsledku (v tomto případě maxima a minima již zpracovaných hodnot).

```
maxmin [1, 2, 4, 3] ⇝* (4, 1)
maxmin [42] ⇝* (42, 42)
```

> **Pan Fešák doporučuje:** Kostru úloh k této kapitole najdete připravené k použití v souboru 05_data.hs v příloze sbírky nebo ve studijních materiálech v ISu.

5.1 Intensionální seznamy

Př. 5.1.1 Reprezentujme seznam účastníků plesu pomocí typu `[(String, Sex)]`, kde první složka dvojice reprezentuje jméno účastníka a druhá složka pohlaví definované následujícím typem:

```
data Sex = Female | Male
```

Pomocí intensionálních seznamů napište funkci

```
allPairs :: [(String, Sex)] -> [(String, String)]
```

která pro daný seznam účastníků vrátí seznam všech možných dvojic účastníků ve tvaru (*muž*, *žena*). Vyzkoušejte si, jak funkci definovat bez jakýchkoli instancí pro datový typ `Sex`. Potom si vyzkoušejte, jaké další možnosti byste měli, pokud bychom přidali instanci `Eq Sex`.

Jak ovlivňuje pořadí generátorů a kvalifikátorů ve vaší definici výsledný seznam a počet kroků potřebných k jeho vygenerování?

```
allPairs [("Jeff", Male), ("Britta", Female),
          ("Annie", Female), ("Troy", Male)] ⇝*
    [("Jeff", "Britta"), ("Jeff", "Annie"),
     ("Troy", "Britta"), ("Troy", "Annie")]
```

Př. 5.1.2 Intensionálním způsobem zapište následující seznamy nebo funkce:

a) `[1, 4, 9, ..., k ^ 2]` (pro pevně dané externě definované `k`)
b) funkci `f`, která ze seznamu seznamů vybere jenom ty delší než 3 prvky
c) `"*****"`
d) `["", "*", "**", "***", ...]`
e) seznam seznamů `[[1], [1, 2], [1, 2, 3], ...]`

Př. 5.1.3 5-seznamy

Intensionálním způsobem zapište výrazy, které se chovají stejně jako následující (předpokládejte externě definované funkce/hodnoty `f`, `p`, `s`, `x`):

a) `map f s`
b) `filter p s`
c) `map f (filter p s)`
d) `repeat x`
e) `replicate n x`
f) `filter p (map f s)`

Př. 5.1.4 Napište funkci, která ze seznamu prvků vygeneruje všechny

a) permutace,
b) variace s opakováním,
c) kombinace.

Prvky ve výsledném seznamu můžou být v libovolném pořadí. Můžete předpokládat, že prvky vstupního seznamu jsou různé. Také se můžete v případě potřeby omezit na seznamy s porovnatelnými prvky (tj. typu `Eq a => a`).

Př. 5.1.5 Která z níže uvedených funkcí je časově efektivnější? Proč? Jak se uvedené funkce chovají pro nekonečné seznamy?

- ```
  f1 :: [a] -> [a]
  f1 s = [ s !! n | n <- [0, 2 .. length s] ]
  ```
- ```
  f2 :: [a] -> [a]
  f2 (x : _ : s) = x : f2 s
  f2 _ = []
  ```

5.2 Lenost, nekonečné datové struktury

Př. 5.2.1 Uvažte nekonečný seznam přirozených čísel `naturals`, který jste definovali v úvodním příkladu 5.η.5. Mějme dále standardní funkci `(!!) :: [a] -> Int -> a` definovanou následovně:

```
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n - 1)
```

Jakou hodnotu má výraz `naturals !! 2`? Ukažte celý výpočet, který k této hodnotě vede. Kde se v tomto výpočtu projeví líná vyhodnocovací strategie?

Vysvětlete, jak by výpočet výrazu `naturals !! 2` probíhal, kdyby Haskell používal *striktní* vyhodnocovací strategii.

Př. 5.2.2 Uvažte opět seznam `naturals` z příkladu 5.η.5. Mějme dále funkci `filter' :: (a -> Bool) -> [a] -> [a]` definovanou následovně:

```
filter' _ [] = []
filter' p (x : xs) = if p x then x : filter' p xs else filter' p xs
```

Jak se bude chovat interpret jazyka Haskell pro vstup `filter' (< 3) naturals`? Ověřte svou hypotézu v interpretu a jeho chování vysvětlete.

Dále uvažte funkci `takeWhile' :: (a -> Bool) -> [a] -> [a]` definovanou následovně:

```
takeWhile' _ [] = []
takeWhile' p (x : xs) = if p x then x : takeWhile' p xs else []
```

Jak se bude chovat interpret jazyka Haskell pro vstup `takeWhile' (< 3) naturals`? Ověřte svou hypotézu v interpretu a jeho chování vysvětlete.

Poznámka: výše uvedené funkce se chovají stejně jako standardní `filter` a `takeWhile`.

Př. 5.2.3 Jaký je význam líného vyhodnocování v následujících výrazech:

a) `let f = f in fst (2, f)`
b) `let f [] = 3 in const True (f [1])`
c) `0 * div 2 0`
d) `snd ("a" * 10, id)`

Př. 5.2.4 Zjistěte, jak se chovají funkce `zip` a `zipWith`, pokud jeden z jejich argumentů je nekonečný seznam.

Uvažte seznam studentů Fakulty informatiky reprezentovaný pomocí seznamu jmen studentů `[String]` tak, že studenti jsou v něm seřazeni podle počtu bodů, které získali z předmětu IB015. Napište funkci `addNumbers :: [String] -> [String]`, která ke každému studentovi přidá jeho pořadí ve vstupním seznamu. Například:

```
addNumbers ["Pablo", "Steve", "Javier", "Gustavo"] ⇝*
```

```
["1. Pablo", "2. Steve", "3. Javier", "4. Gustavo"]
```

Zkuste funkci `addNumbers` naprogramovat tak, aby vstupní seznam prošla právě jednou (tedy zejména nepoužívejte funkci `length`).

> **Pan Fešák doporučuje:** Vzpomeňte si na funkci `show :: Show a => a -> String`.

Př. 5.2.5 S pomocí intensionálního seznamu definujte nekonečný seznam `integers :: [Integer]`, který obsahuje právě všechna celá čísla. Seznam `integers` musí splňovat, že každé celé číslo v něm jde vygenerovat po konečném počtu kroků. Jinými slovy, pro každé celé číslo `z` musí existovat `i` takové, že `(integers !! i) == z`.

Př. 5.2.6 Definujte nekonečný seznam `threeSum :: [(Integer, Integer, Integer)]`, který obsahuje právě ty trojice kladných čísel `(x, y, z)`, pro které platí `x + y == z`. Seznam `threeSum` musí splňovat, že každá taková trojice v něm jde vygenerovat po konečném počtu kroků. Jinými slovy, pro každou trojici `(x, y, z)`, kde `x + y == z`, musí existovat `i` takové, že `(threeSum !! i)` $\rightsquigarrow^*$ `(x, y, z)`.

Př. 5.2.7 Uvažte libovolný výraz a počet kroků vyhodnocení tohoto výrazu při použití líné vyhodnocovací strategie a počet kroků při použití striktní vyhodnocovací strategie. Jaký je obecně vztah ($=, <, \leq, >, \geq$, ani jedno) mezi těmito počty kroků? Jaký je obecně vztah mezi počty kroků normální vyhodnocovací strategie a striktní vyhodnocovací strategie?

Př. 5.2.8 Jaké jsou výhody líné vyhodnocovací strategie? Jaké jsou výhody striktní vyhodnocovací strategie?

Př. 5.2.9 Pomocí některé z funkcí `iterate`, `repeat`, `replicate`, `cycle` vyjádřete nekonečné seznamy:

a) Seznam nekonečně mnoha hodnot `True`.
b) Rostoucí seznam všech mocnin čísla 2.
c) Rostoucí seznam všech mocnin čísla 3 na sudý exponent.
d) Rostoucí seznam všech mocnin čísla 3 na lichý exponent.
e) Alternující seznam `-1` a `1`: `[1, -1, 1, -1, ...]`.
f) Seznam řetězců `["", "*", "**", "***", "****", ...]`.
g) Seznam zbytků po dělení `4` pro seznam `[1 ..]`: `[1, 2, 3, 0, 1, 2, 3, 0, ...]`.

Př. 5.2.10 Naprogramujte funkci `differences :: [Integer] -> [Integer]`, která pro nekonečný seznam `[x1, x2, x3,...]` vypočítá seznam rozdílů po sobě jdoucích dvojic prvků, tedy seznam `[(x2 - x1), (x3 - x2), (x4 - x3), ...]`.

Zkuste funkci `differences` naprogramovat bez explicitního použití rekurze, pomocí funkcí `zipWith` a `tail`.

Př. 5.2.11 Naprogramujte funkci `values :: (Integer -> a) -> [a]`, která pro zadanou funkci `f :: Integer -> a` vypočítá nekonečný seznam jejích hodnot `[f 0, f 1, f 2, ...]`.

Uvažte funkci `differences` z předchozí úlohy.

a) Čemu odpovídá seznam `differences (values f)`?
b) Čemu odpovídá seznam `differences (differences (values f))`?

Pomocí funkcí `values`, `differences`, `zip3` a dalších vhodných funkcí na seznamech napište funkci `localMinima :: (Integer -> Integer) -> [Integer]`, která pro zadanou funkci `f :: Integer -> Integer` vypočítá seznam hodnot, ve kterých funkce `f` nabývá na kladných vstupech lokálního minima (tedy hodnot `f n` takových, že `n > 0`, `f (n - 1) > f n`

a zároveň `f (n + 1) > f n`).

Př. 5.2.12  Definujte Fibonacciho posloupnost, tj. seznam kladných celých čísel `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...]`. Můžete ji definovat jako seznam hodnot (typ `[Integer]`) nebo jako funkci, která vrátí konkrétní Fibonacciho číslo (`Integer -> Integer`). Jaká je ve Vaší implementaci složitost výpočtu n-tého čísla Fibonacciho posloupnosti?

Př. 5.2.13   Pomocí rekurzivní definice a funkce `zipWith` vyjádřete Fibonacciho posloupnost tak, že výpočet každého dalšího prvku proběhl v konstantním čase (tj. počet výpočetních kroků nutný k získání dalšího čísla posloupnosti není nijak závislý na tom, kolikáté číslo to je).

Př. 5.2.14  Protože možnost definovat nekonečné seznamy není žádná magie, ale vyplývá z vlastností vyhodnocovací strategie jazyka Haskell, není překvapivé, že lze definovat nekonečné hodnoty i pro jiné vlastní datové typy. Vzpomeňte si na definici datového typu binárních stromů:

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

Na rozdíl od nekonečných seznamů bohužel není zaručené, že při výpisu nekonečného stromu v interpretu uvidíte dříve nebo později každý jeho prvek. Výpis totiž probíhá *do hloubky*, a tudíž se nejprve vypisuje celá nejlevější větev stromu. Ta však nemusí být konečná, takže se výpis nemusí dostat k ostatním větvím stromu.

Pro další práci s nekonečnými binárními stromy se tedy bude hodit nejprve definovat funkci, která pro zadaný potenciálně nekonečný strom vrátí jeho konečnou část, kterou lze vypsat celou. Definujte tedy funkci `treeTrim :: BinTree a -> Integer -> BinTree a` takovou, že pro zadaný strom `t` a hloubku `h` bude výsledkem `treeTrim t h` konečný strom, který obsahuje ty uzly stromu `t`, které jsou nejvýše v hloubce `h`. Vzpomeňte si, že kořen stromu má hloubku 0. Poté definujte:

a) funkci `treeRepeat :: a -> BinTree a` jako analogii seznamové funkce `repeat`. Funkce tedy vytvoří nekonečný strom, který má v každém uzlu zadanou hodnotu. Tedy například

```
treeTrim (treeRepeat 42) 1 ⇝*
    Node 42 (Node 42 Empty Empty) (Node 42 Empty Empty)
```

b) funkci `treeIterate :: (a -> a) -> (a -> a) -> a -> BinTree a` jako analogii seznamové funkce `iterate`. Levý potomek každého uzlu bude mít hodnotu vzniklou aplikací první zadané funkce a pravý aplikací druhé zadané funkce. Tedy například

```
treeTrim (treeIterate (+1) (*4) 2) 1 ⇝*
    Node 2 (Node 3 Empty Empty) (Node 8 Empty Empty)
```

c) pomocí funkce `treeIterate` vyjádřete nekonečný binární strom `depthTree` typu `BinTree Integer`, jehož každý uzel v sobě obsahuje svou hloubku. Tedy například

```
treeTrim depthTree 1 ⇝*
    Node 0 (Node 1 Empty Empty) (Node 1 Empty Empty)
```

Př. 5.2.15 Definujte nějaký binární strom, který obsahuje alespoň jednu nekonečnou větev a alespoň jednu konečnou větev.

Př. 5.2.16  Definujte nekonečný seznam `nonNegativePairs :: [(Integer, Integer)]`, který obsahuje právě všechny dvojice kladných čísel `(x, y)`. Seznam `nonNegativePairs` musí splňovat, že každá dvojice kladných čísel v něm jde vygenerovat po konečném počtu kroků.

Př. 5.2.17  Definujte nekonečný seznam `positiveLists :: [[Integer]]`, který obsahuje právě všechny konečné seznamy kladných čísel. Seznam `positiveLists` musí splňovat, že každý konečný seznam kladných čísel v něm jde vygenerovat po konečném počtu kroků.

5.3 Akumulační funkce na seznamech

Jindřiška upozorňuje: S akumulačními funkcemi na seznamech se můžeme setkat i ve většině imperativních programovacích jazyků. Například v Pythonu je to funkce `functools.reduce`, v C++ `std::accumulate` a v C# metoda `Aggregate`. V drtivé většině případů tyto funkce zpracovávají data ve stejném pořadí jako funkce `foldl`.

Pan Fešák doplňuje: Pokud si necháme v interpretru otypovat například `foldr`, dozvíme se obecnější typ, než nám relativně čitelný `foldr :: (a -> b -> b) -> b -> [a] -> b`. Je to dáno tím, že foldy jde v Haskellu používat i na jiné datové struktury, než je seznam. Většina těchto datových struktur (nebo jejich fungování s foldy) je však mimo rozsah tohoto kurzu, nám tedy bude stačit si mentálně v typech nahradit `Foldable` za seznamy, například u `foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b` vidíme, že `t` je `Foldable` a tedy `t a` můžeme pro naše účely zjednodušit na `[a]`.

Připomeňme si, jak můžeme akumulační funkce definovat (definice v knihovně se na seznamech chovají stejně, a to včetně chování k lenosti, mají ale obecnější typ a mohou být trochu optimalizované):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ base []     = base
foldr f base (x:xs) = f x (foldr f base xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ []     = error "foldr1: empty list"
foldr1 _ [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 _ []     = error "foldl1: empty list"
foldl1 f (x:xs) = foldl f x xs
```

Př. 5.3.1  Definujte následující funkce rekurzivně:

a) `product'` – součin prvků seznamu
b) `length'` – počet prvků seznamu

c) `map'` – funkci `map`

Co mají tyto definice společného? Jak by vypadalo jejich zobecnění (tj. funkce, pomocí které se použitím vhodných argumentů dají všechny tyto tři funkce implementovat)?

Př. 5.3.2 5-fold »=

Pomocí vhodné akumulační funkce `foldr`, `foldl`, `foldr1` nebo `foldl1` implementujte následující funkce. Jinými slovy, definice musí být jediný řádek tvaru `funkce argumenty = fold* ...`, tedy speciálně nesmíte samostatně ošetřovat prázdný seznam.

> **Jindřiška varuje:** Varianty `foldr1` a `foldl1` jsou použitelné jen pokud nepotřebujete, aby funkce fungovala na prázdném seznamu. Tento případ však chceme pokrýt kdykoli to rozumně lze.

a) Funkci `sumFold`, která vrátí součet čísel v zadaném seznamu.

```
sumFold :: Num a => [a] -> a
sumFold [1, 2, 4, 5, 7, 6, 2] ⇝* 27
```

b) Funkci `productFold`, která vrátí součin čísel v zadaném seznamu.

```
productFold :: Num a => [a] -> a
productFold [1, 2, 4, 0, 7, 6, 2] ⇝* 0
```

c) Funkci `orFold`, která vrátí `True`, pokud se v zadaném seznamu nachází aspoň jednou hodnota `True`, jinak vrátí `False`.

```
orFold :: [Bool] -> Bool
orFold [False, True, False] ⇝* True
```

d) Funkci `lengthFold`, která vrátí délku zadaného seznamu.

```
lengthFold :: [a] -> Int
lengthFold ["Holographic Rick", "Shrimp Rick", "Wasp Rick"] ⇝* 3
```

e) Funkci `maximumFold`, která vrátí maximální prvek ze zadaného neprázdného seznamu.

```
maximumFold :: Ord a => [a] -> a
maximumFold "patrick star" ⇝* 't'
```

Př. 5.3.3

Všechny funkce z předchozího příkladu již jsou ve standardní knihovně jazyka Haskell implementované. Pomocí dokumentace zjistěte, jak se ve standardní knihovně jmenují.

Př. 5.3.4 »=

Bez použití interpretru určete, jak se vyhodnotí akumulační funkce s následujícími hodnotami a zda vyhodnocování skončí. Následně si chování určete v interpretru, dávejte ale pozor, abyste případné nekončící výpočty včas zabili (ctrl+c) aby vám nedošla paměť.

a) `foldr (-) 0 [1, 2, 3]`
b) `foldl (-) 0 [1, 2, 3]`
c) `foldr (&&) True (True : False : repeat True)`
d) `foldl (&&) True (True : False : repeat True)`

> **Pan Fešák připomíná:** Funkce `foldr` je takzvaný *katamorfismus* na seznamech.

Př. 5.3.5

Definujte funkci `subtractlist`, která odečte druhý a všechny další prvky *neprázdného* seznamu od jeho prvního prvku, tj. `subtractlist [x1, x2, ..., xn]` se vyhodnotí na `x1 - x2 - ... - xn`.

Př. 5.3.6 Uvažme funkci: `foldr (.) id`

a) Jaký je význam této funkce?
b) Jaký je její typ?
c) Uveďte příklad částečné aplikace této funkce na jeden argument.
d) Uveďte příklad úplné aplikace této funkce na kompletní seznam argumentů.

Př. 5.3.7 S použitím vhodné akumulační funkce definujte funkci `append' :: [a] -> [a] -> [a]`, která vypočítá zřetězení vstupních seznamů. Tedy funkce `append'` se bude chovat stejně jako knihovní funkce `(++)`.

Zkuste všechny výrazy použité při definici funkce `append'` co nejvíce η-redukovat.

Př. 5.3.8 S použitím vhodné akumulační funkce definujte funkci `reverse' :: [a] -> [a]`, která s *lineární* časovou složitostí otočí vstupní seznam (dejte si pozor na to, že operátor `++` má lineární časovou složitost vzhledem k délce prvního argumentu).

Př. 5.3.9 5-fold

Vaší úlohou je implementovat funkci dle specifikace v zadání za použití standardních funkcí `foldr`, `foldl`, `foldr1`, `foldl1`. Pokud není řečeno jinak, řešení by nemělo obsahovat formální parametry – má tedy být v následujícím tvaru:

```
functionName = foldr (function) (term)
```

Jestliže je možné příklad řešit více než jednou z nabízených akumulačních funkcí, vyberte tu, která je nejefektivnější. K většině zadání je dostupný i ukázkový výsledek na jednom seznamu sloužící jako ilustrace.

> **Jindřiška varuje:** Každé řešení zapisujte i s typem funkce, v některých příkladech by bez něj nemuselo jít zkompilovat (důvod je poněkud složitější).

a) Funkce `concatFold` vrátí zřetězení prvků zadaného seznamu seznamů.

```
concatFold :: [[a]] -> [a]
concatFold ["pineapple", "apple", "pen"] ⇝* "pineappleapplepen"
```

b) Funkce `listifyFold` nahradí každý prvek jednoprvkovým seznamem s původním prvkem.

```
listifyFold :: [a] -> [[a]]
listifyFold [1, 3, 4, 5] ⇝* [[1], [3], [4], [5]]
```

c) Funkce `nullFold` vrátí `True`, pokud je zadaný seznam prázdný, jinak vrátí `False`.

```
nullFold :: [a] -> Bool
nullFold ["Aang", "Appa", "Momo", "Zuko"] ⇝* False
```

d) Funkce `composeFold` vezme seznam funkcí a hodnotu, a vrátí hodnotu, která vznikne postupným aplikováním funkcí v seznamu na danou hodnotu (poslední funkce se aplikuje jako první, první jako poslední).

```
composeFold :: [a -> a] -> a -> a
composeFold [(* 4),(+ 2)] 3 ⇝* 20
```

e) Funkce `idFold` vrátí zadaný seznam beze změny.

```
idFold :: [a] -> [a]
idFold ["Lorelai", "Rory", "Luke"] ⇝* ["Lorelai", "Rory", "Luke"]
```

f) Funkce `mapFold` vezme funkci a seznam, a vrátí seznam, který vznikne aplikací zadané funkce na každý prvek zadaného seznamu. Pro funkci použijte formální argument.

```
mapFold :: (a -> b) -> [a] -> [b]
mapFold (+ 5) [1, 2, 3] ⇝* [6, 7, 8]
```

g) Funkce `headFold` vrátí první prvek zadaného neprázdného seznamu.

```
headFold :: [a] -> a
headFold ["Light", "L", "Ryuk", "Misa"] ⇝* "Light"
```

h) Funkce `lastFold` vrátí poslední prvek zadaného neprázdného seznamu.

```
lastFold :: [a] -> a
lastFold ["Edward", "Alphonse", "Winry", "Mustang"] ⇝* "Mustang"
```

i) Funkce `maxminFold` vrátí maximální a minimální prvek ze zadaného neprázdného seznamu ve formě uspořádané dvojice. V definici použijte i formální argument funkce `maxminFold`, bude se Vám hodit. Funkce by měla projít zadaný seznam pouze jednou! Vzpomeňte též na příklad 5.η.6, může vám dát nápovědu, jak problém vyřešit.

```
maxminFold :: Ord a => [a] -> (a, a)
maxminFold ["Lana", "Sterling", "Cyril"] ⇝* ("Sterling", "Cyril")
```

j) Funkce `suffixFold` vrátí seznam všech přípon zadaného seznamu (jako první bude samotný seznam, poslední bude prázdný seznam).

```
suffixFold :: [a] -> [[a]]
suffixFold "abcd" ⇝* ["abcd", "bcd", "cd", "d", ""]
```

k) Funkce `filterFold` vezme predikát a seznam a vrátí seznam, který vznikne ze zadaného seznamu vyloučením všech prvků, na kterých predikát vrátí `False`. Pro predikát použijte formální argument.

```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold odd [1, 2, 4, 8, 6, 2, 5, 1, 3] ⇝* [1, 5, 1, 3]
```

l) Funkce `oddEvenFold` vrátí v uspořádané dvojici seznamy prvků z lichých a sudých pozic původního seznamu.

```
oddEvenFold :: [a] -> ([a], [a])
oddEvenFold [1, 2, 7, 5, 4] ⇝* ([1, 7, 4], [2, 5])
```

m) Funkce `takeWhileFold` vezme predikát a seznam, a vrátí nejdelší prefix seznamu, pro jehož každý prvek vrátí predikát hodnotu `True`. Pro predikát použijte formální argument.

```
takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold even [2, 4, 1, 2, 4, 5, 8, 6, 8] ⇝* [2, 4]
```

n) Funkce `dropWhileFold` vezme predikát a seznam, a vrátí zadaný seznam bez nejdelšího prefixu, pro jehož každý prvek vrátí predikát hodnotu `True`. Pro predikát použijte formální argument.

```
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold odd [1, 2, 5, 9, 1, 7, 4, 6] ⇝* [2, 5, 9, 1, 7, 4, 6]
```

Př. 5.3.10 Definujte funkci `foldl` pomocí funkce `foldr`.

Př. 5.3.11 Naprogramujte funkci `insert :: Ord a => a -> [a] -> [a]` takovou, že výsledkem vyhodnocení `insert x xs` pro uspořádaný seznam `xs` bude uspořádaný seznam, který vznikne vložením prvku `x` na vhodné místo v seznamu `xs`. Například tedy:

`insert 5 [1, 3, 18, 19, 30]` $\rightsquigarrow^*$ `[1, 3, 5, 18, 19, 30]`

Funkci `insert` nemusíte implementovat pomocí akumulačních funkcí, avšak chcete-li si je procvičit, můžete.

Pomocí funkce `insert` a vhodné akumulační funkce naprogramujte funkci `insertSort :: Ord a => [a] -> [a]`, která seřadí vstupní seznam pomocí algoritmu *řazení vkládáním (insert sort)*.

Př. 5.3.12 Proč je implementace funkce `or` (logická disjunkce všech hodnot v seznamu) pomocí funkce `foldr` lepší než pomocí `foldl`?

Př. 5.3.13 Pomocí dokumentace a internetu zjistěte, co dělají funkce `foldl'` a `foldl1'` z modulu `Data.List`. Jak se liší od funkcí `foldl` a `foldl1`? Kdy byste použili funkci `foldl'` místo funkce `foldl`? Zamyslete se, proč knihovna neobsahuje funkci `foldr'`.

Př. 5.3.14 Je možné definovat funkci `f` tak, aby se `foldl f [] s` vyhodnotilo na seznam obsahující jenom prvky ze sudých míst v seznamu `s`?

Je možné definovat takovou funkci ve tvaru `foldEveryOther s = snd (foldl f v s)` pro vhodné hodnoty `f` a `v`?

Př. 5.3.15 Mějme funkci `foldr2` definovanou následovně:

```
foldr2 :: (a -> a -> b -> b) -> (a -> b) -> b -> [a] -> b
foldr2 f2 f1 f0 [] = f0
foldr2 f2 f1 f0 [x] = f1 x
foldr2 f2 f1 f0 (x : y : s) = f2 x y (foldr2 f2 f1 f0 s)
```

Zkuste definovat funkci `foldr` pomocí `foldr2` a funkci `foldr2` pomocí `foldr`, nebo zdůvodněte, proč to není možné.

5.4 Akumulační funkce na vlastních datových typech

> **Pan Fešák doplňuje:** V tomto případě myslíme takzvané *katamorfismy*, tedy funkce, které nahrazují výskyty příslušných hodnotových konstruktorů za funkce dané arity. Katamorfismy jsou něco jiného než instance typové třídy `Foldable` pro daný datový typ (těmi se tu v podstatě zabývat nebudeme, najít je můžete jedině v příkladu 5.4.7, který je výrazně nad rámec předmětu).

Př. 5.4.1 Mějme klasický datový typ `BinTree a` reprezentující binární stromy, které mají v uzlech hodnoty typu `a`:

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

Definujte funkci `treeFold`, která bude analogií seznamové funkce `foldr`, tedy tzv. *katamorfismem* na typu `BinTree`. Tedy volání `treeFold n e t` nahradí ve stromě `t` všechny hodnotové konstruktory `Node` funkcí `n` a všechny hodnotové konstruktory `Empty` hodnotou `e`. Například chceme, aby

- funkce `treeFold (\v resultL resultR -> v + resultL + resultR) 0` sečetla všechna čísla v zadaném stromě,

- funkce `treeFold (\v resultL resultR -> v * resultL * resultR) 1` vynásobila všechna čísla v zadaném stromě a
- funkce `treeFold (\v resultL resultR -> v || resultL || resultR) False` rozhodla, jestli v zadaném stromě je alespoň jedna hodnota `True`.

Zkuste si před vlastní implementací rozmyslet, jaký má funkce `treeFold` mít typ.

Př. 5.4.2 5-treeFold

Vaší úlohou je implementovat pomocí funkce `treeFold` z předchozí úlohy několik funkcí, které pracují se stromy typu `BinTree a`. Jestli úloha neříká jinak, řešení by mělo být bez formálních parametrů, tedy v následujícím tvaru:

```
functionName = treeFold (function) (term)
```

Definici datového typu `BinTree a`, několika testovacích stromů a funkce `treeFold` naleznete v souboru 05_treeFold.hs (dá se stáhnout i z ISu).

Ke většině úloh je dostupný i ukázkový výsledek na předem zvoleném stromě (slouží jako ilustrace, co zadání vlastně požaduje). Kvůli přehlednosti jsou ukázkové stromy pojmenované a jejich definice najdete až za poslední podúlohou.

a) Funkce `treeSize` vrátí počet uzlů v zadaném stromě.

```
treeSize :: BinTree a -> Int
treeSize tree01 ⇝* 6
treeSize tree06 ⇝* 5
```

b) Funkce `treeHeight` vrátí výšku zadaného stromu (poznámka: prázdný strom má výšku 0, jednouzlový strom má výšku 1).

```
treeHeight :: BinTree a -> Int
treeHeight tree03 ⇝* 2
treeHeight tree01 ⇝* 3
```

c) Funkce `treeList` vrátí seznam hodnot ze všech uzlů. Nejdříve uveďte hodnoty z levého podstromu, pak hodnotu v uzlu a následně hodnoty z pravého podstromu (tzv. *inorder* procházení stromu).

```
treeList :: BinTree a -> [a]
treeList tree01 ⇝* [5, 3, 2, 1, 4, 1]
treeList tree02 ⇝* ["A", "B", "C", "D", "E"]
```

d) Funkce `treeConcat` vrátí zřetězení hodnot ze všech uzlů.

```
treeConcat :: BinTree [a] -> [a]
treeConcat tree02 ⇝* "ABCDE"
```

e) Funkce `treeMax` vrátí maximální hodnotu ze všech hodnot v uzlech. Hodnoty musí být z typové třídy `Ord` a `Bounded` (poznámka: zkuste hodnoty `minBound` a `maxBound`). Upozornění: Stromy, které budete používat na vyhodnocování mějte explicitně otypovány, jinak můžete narazit na problém při kompilaci (důvod je poněkud složitější).

```
treeMax :: (Ord a, Bounded a) => BinTree a -> a
treeMax tree01 ⇝* 5
treeMax tree03 ⇝* (3, 3)
```

f) Funkce `treeFlip` vrátí zadaný strom, avšak každá jeho pravá větev bude vyměněna s příslušnou levou větví.

```
treeFlip :: BinTree a -> BinTree a
treeFlip tree01 ⇝*
    Node 2 (Node 4 (Node 1 Empty Empty)
```

```
    (Node 1 Empty Empty)) (Node 3 Empty (Node 5 Empty Empty))
treeConcat (treeFlip tree02) ⇝* "EDCBA"
```

g) Funkce `treeId` vrátí zadaný strom v nezměněné podobě (Pozor! Stále vyžadujeme použití funkce `treeFold`!).

```
treeId :: BinTree a -> BinTree a
treeId tree05 ⇝* tree05
```

h) Funkce `rightMostBranch` vrátí seznam hodnot nejpravější větve zadaného stromu (v nejpravější větvi nikdy „nezatáčíme doleva“).

```
rightMostBranch :: BinTree a -> [a]
rightMostBranch tree01 ⇝* [2, 4, 1]
rightMostBranch tree02 ⇝* ["C", "E"]
```

i) Funkce `treeRoot` vrátí kořenový prvek zadaného stromu. Jestli je strom prázdný, program havaruje (poznámka: můžete použít hodnotu `undefined`).

```
treeRoot :: BinTree a -> a
treeRoot tree01 ⇝* 2
```

j) Funkce `treeNull` zjistí, jestli je zadaný strom prázdný (podobá se funkci `null` pro seznamy).

```
treeNull :: BinTree a -> Bool
treeNull tree01 ⇝* False
treeNull tree04 ⇝* True
```

k) Funkce `leavesCount` vrátí počet listů v zadaném stromě (list je každý uzel, který nemá potomky).

```
leavesCount :: BinTree a -> Int
leavesCount tree01 ⇝* 3
leavesCount tree04 ⇝* 0
```

l) Funkce `leavesList` vrátí seznam hodnot z listů zadaného stromu. Preferované pořadí listů v seznamu je zleva doprava.

```
leavesList :: BinTree a -> [a]
leavesList tree01 ⇝* [5, 1, 1]
leavesList tree02 ⇝* ["B", "D"]
```

m) Funkce `treeMap` aplikuje zadanou funkci na hodnotu v každém uzlu zadaného stromu (poznámka: funkce pracuje podobně jako `map` na seznamech). Výsledná funkce může mít jeden formální parametr.

```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMax (treeMap negate tree01) ⇝* -1
```

n) Funkce `treeAny` zjistí, jestli alespoň jedna hodnota v zadaném stromě splňuje zadaný predikát (tedy se na něm vyhodnotí na `True`). Výsledná funkce může mít jeden formální parametr.

```
treeAny :: (a -> Bool) -> BinTree a -> Bool
treeAny (==10) tree01 ⇝* False
treeAny even tree01 ⇝* True
treeAny null tree02 ⇝* False
```

o) Funkce `treePair` zjistí, jestli je v každém uzlu stromu první složka uspořádané dvojice rovná druhé složce této dvojice.

```
treePair :: Eq a => BinTree (a,a) -> Bool
treePair tree03 ⇝* False
```

p) Funkce `subtreeSums` vloží do každého uzlu zadaného stromu součet všech uzlů podstromu určeného tímto uzlem.

```
subtreeSums :: Num a => BinTree a -> BinTree a
subtreeSums tree01 ⇝* Node 16 (Node 8 (Node 5 Empty Empty) Empty)
    (Node 6 (Node 1 Empty Empty) (Node 1 Empty Empty))
```

Př. 5.4.3 Mějme klasický datový typ `RoseTree a` reprezentující stromy libovolné arity, které mají v uzlech hodnoty typu `a`:

```
data RoseTree a = RoseNode a [RoseTree a]
                deriving Show
```

Definujte funkci `roseTreeFold`, která bude analogií seznamové funkce `foldr`, tedy tzv. *katamorfismem* na typu `RoseTree`. Tedy volání `roseTreeFold n e t` nahradí ve stromě `t` všechny hodnotové konstruktory `RoseNode` funkcí `n`. Například chceme, aby

- funkce `roseTreeFold (\v sums -> v + sum sums)` sečetla všechna čísla v zadaném stromě,
- funkce `roseTreeFold (\v products -> v * product products)` vynásobila veškerá čísla v zadaném stromě a
- funkce `roseTreeFold (\v ors -> v || or ors)` rozhodla, jestli v zadaném stromě je alespoň jedna hodnota `True`.

Zkuste si před vlastní implementací rozmyslet, jaký má funkce `roseTreeFold` mít typ.

Př. 5.4.4 Uvažte datový typ `RoseTree a` a akumulační funkci `roseTreeFold` z předchozí úlohy.

Pomocí funkce `roseTreeFold` implementujte analogie všech funkcí z úlohy 5.4.2, které ale budou tentokrát pracovat se stromy libovolné arity.

Př. 5.4.5 Mějme datový typ `Nat` reprezentující přirozená čísla:

```
data Nat = Succ Nat | Zero deriving (Eq, Show)
```

Definujte funkci `natFold` typu `(a -> a) -> a -> Nat -> a`, která je tzv. *katamorfismem* na typu `Nat`. Jinými slovy funkce `natFold` nahrazuje všechny hodnotové konstruktory datového typu, podobně jako funkce `foldr`, `treeFold` a `roseTreeFold`.

Příklady zamýšleného použití funkce `nfold`:

a) Funkce `natFold (Succ . Succ) Zero :: Nat -> Nat` zdvojnásobuje hodnotu přirozeného čísla typu `Nat`.

b) Funkce `natFold (1 +) 0 :: Nat -> Int` převádí hodnotu typu `Nat` do celých čísel typu `Int`.

Př. 5.4.6 Pomocí funkce `natFold` z minulého příkladu naprogramujte

a) funkci, která sečte dvě přirozená čísla typu `Nat`,

b) funkci, která rozhodne, jestli je zadané přirozené číslo typu `Nat` sudé a

c) funkci, která vynásobí dvě přirozená čísla typu `Nat` (tady se Vám možná bude hodit některá z již naprogramovaných funkcí).

Všechny předchozí funkce zkuste naprogramovat bez převodu přirozeného čísla typu `Nat` na celé číslo typu `Int` nebo `Integer`.

Př. 5.4.7 Nastudujte si, co je minimální implementace `Foldable` a implementujte instance `Foldable BinTree` pro náš typ binárních stromů (jedná se o instanci pro typový konstruktor, nikoli pro konkrétní či polymorfní typ, za `BinTree` tedy v tomto případě nemá být parametr). Následně si ověřte, že standardní funkce `sum`, `product`, `any`, `all`, apod. fungují nyní pro `BinTree a` (s vhodným `a`).

```haskell
tree123 :: Num a => BinTree a
tree123 = Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)

sum tree123 ⇝* 6
any odd tree123 ⇝* True
all odd tree123 ⇝* False
foldr (:) [] tree123 ⇝* [1, 2, 3]

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

⁂

Na konci pátého cvičení byste měli umět:

- umět použít intensionální seznamy pro generování nových seznamů ze zadaných seznamů;
- poznat, kdy se dá zadaný problém vyřešit pomocí intensionálních seznamů;
- umět pomocí intensionálních seznamů definovat nekonečné seznamy;
- použít líné vyhodnocování k práci s nekonečnými seznamy;
- umět použít nekonečné seznamy v praxi;
- použít akumulační funkce na jednoduché operace na seznamech jako součet všech prvků, maximum seznamu a podobně;
- poznat, kdy je možné problém jednoduše vyřešit pomocí akumulačních funkcí a také vybrat akumulační funkci, která pro tento účel bude vhodná.

Řešení

Řeš. 5.η.1
```
divisors :: Integer -> [Integer]
divisors n = [ x | x <- [1 .. n], mod n x == 0 ]
```

Řeš. 5.η.4

a)
```
countPassed :: [(UCO, [String])] -> [(UCO, Int)]
countPassed m = [ (uco, length passed) | (uco, passed) <- m ]
```

b)
```
atLeastTwo :: [(UCO, [String])] -> [UCO]
atLeastTwo m = [ uco | (uco, passed) <- m, length passed >= 2 ]
```

Nebo alternativně s využitím vzorů:

```
atLeastTwo' :: [(UCO, [String])] -> [UCO]
atLeastTwo' m = [ uco | (uco, x:y:_) <- m ]
```

c)
```
passedIB015 :: [(UCO, [String])] -> [UCO]
passedIB015 m = [ uco | (uco, passed) <- m, elem "IB015" passed ]
```

d)
```
passedBySomeone :: [(UCO, [String])] -> [String]
passedBySomeone m = [ code | (_, passed) <- m, code <- passed ]
```

Řeš. 5.η.5 Funkci `naturalsFrom` jde definovat pomocí jednoduché rekurzivní definice, která nebude mít žádný bázový případ.

```
naturalsFrom :: Integer -> [Integer]
naturalsFrom n = n : naturalsFrom (n + 1)

naturals :: [Integer]
naturals = naturalsFrom 0
```

> **Pan Ešák upřesňuje:** Protože definici funkce `naturalsFrom` chybí bázový případ a její volání nikdy neskončí, tato definice není ve skutečnosti korektní rekurzivní definice v matematickém slova smyslu. Přesněji řečeno je tato definice *korekurzivní* (viz https://en.wikipedia.org/wiki/Corecursion).

Alternativně lze řešit pomocí enumeračního zápisu:

```
naturalsFrom' :: Integer -> [Integer]
naturalsFrom' n = [n..]
```

Řeš. 5.η.6
```
maxmin :: Ord a => [a] -> (a, a)
maxmin (x:xs) = maxmin' (x, x) xs
  where
    maxmin' res [] = res
    maxmin' (ma, mi) (a:as) = maxmin' (ma `max` a, mi `min` a) as
```

Řeš. 5.1.1 Elegantní možností je využít vzory s konkrétními hodnotami pro pohlaví:

```
allPairs ppl = [ (m, f) | (m, Male) <- ppl, (f, Female) <- ppl ]
```

Toto řešení využívá toho, že položky generátory (výrazu `... <- [...]`), které nesplňují vzor se automaticky vynechají. Do proměnné `m` se tak dosazují pouze ty hodnoty, které jsou ve dvojici s `Male`, zatímco do `f` se dosazují pouze ty, které jsou ve dvojici s `Female`.

Pokud bychom navíc měli instanci `Eq Sex`, mohli bychom využít i následujících řešení:

```
allPairs' people = [ (m, f) | (m, isM) <- people, isM == Male,
                              (f, isF) <- people, isF == Female ]
```

Jiným funkčním řešením by bylo například:

```
allPairs'' people = [ (m, f) | (m, isM) <- people, (f, isF) <- people,
                               isM == Male, isF == Female]
```

To je ale zbytečně neefektivní: ke každému `m` se postupně zkusí všechna možná `f` i tehdy, když `m` není muž a proto stejně takové přiřazení vzápětí zahodíme. Kvalifikátory tedy chceme mít co nejvíce vlevo to jde, aby se nevyhovující přiřazení vyloučila co nejdříve. Vzájemné pořadí generátorů pak ovlivňuje, zda jsou ve výsledném seznamu shlukovány páry podle mužů, nebo žen.

Řeš. 5.1.2

a)
```
[ x^2 | x <- [1 .. k] ]
```

b)
```
f :: [[a]] -> [[a]]
f s = [ t | t <- s, length t > 3 ]
```

c)
```
[ '*' | _ <- [1 .. 5] ]
```

d)
```
[ ['*' | _ <- [1 .. n]] | n <- [0 ..] ]
```

e)
```
[ [1 .. n] | n <- [1 ..] ]
```

Řeš. 5.1.3

a)
```
[ f x | x <- s ]
```

b)
```
[ x | x <- s, p x ]
```

c)
```
[ f x | x <- s, p x ]
```

d)
```
[ x | _ <- [1 ..] ]
```

e)
```
[ x | _ <- [1 .. n] ]
```

f)
```
[ x | t <- s, let x = f t, p x ]
```

případně `[ f x | x <- s, p (f x) ]`, ale toto řešení je méně efektivní

Řeš. 5.1.4

a)
```
perm :: Eq a => [a] -> [[a]]
perm [] = [[]]
perm s  = [m : n | m <- s, n <- perm (filter (m /=) s)]
```

b)
```
varrep :: Int -> [a] -> [[a]]
varrep 0 s = [[]]
varrep k s = [m : n | m <- s, n <- varrep (k - 1) s]
```

c)
```
comb :: Int -> [a] -> [[a]]
comb 0 _   = [[]]
comb k xs0 =
    [m : t | (m, n) <- zip xs0 . tails . tail $ xs0, t <- comb (k - 1)
     ↪ n]
      where
          tails []       = [[]]
          tails (x : xs) = (x : xs) : tails xs
```

Tady lze případně použít funkci `tails` z modulu `Data.List`, viz https://haskell.fi.muni.cz/doc/base/Data-List.html#v:tails.

Řeš. 5.1.5 Nechť $n =$ `length s`. Lepší časovou složitost má funkce `f2`, protože projde seznamem jenom jednou, tedy má lineární časovou složitost. Na druhé straně `f1` vykoná nejvíce $n/2 + 1$ volání funkce `(!!)`. Tato volání se v tomto případě vykonají každé v lineárním čase vzhledem k druhému argumentu funkce `(!!)`. Dohromady tedy vyhodnocení funkce `f1` vyžaduje kvadratický čas.

Řeš. 5.2.1

```
naturals !! 2
  ⇝ naturalsFrom 0 !! 2
  ⇝ (0 : naturalsFrom (0 + 1)) !! 2
  ⇝ naturalsFrom (0 + 1) !! (2 - 1)
  ⇝ ((0 + 1) : naturalsFrom ((0 + 1) + 1)) !! (2 - 1)
  ⇝ ((0 + 1) : naturalsFrom ((0 + 1) + 1)) !! 1
  ⇝ naturalsFrom ((0 + 1) + 1) !! (1 - 1)
  ⇝ (((0 + 1) + 1) : naturalsFrom (((0 + 1) + 1) + 1)) !! (1 - 1)
  ⇝ (((0 + 1) + 1) : naturalsFrom (((0 + 1) + 1) + 1)) !! 0
  ⇝ (0 + 1) + 1
  ⇝ 1 + 1
  ⇝ 2
```

Nejdůležitější projev lenosti je vidět v předposledním řádku: zahození (a tedy nevyhodnocení) podvýrazu `naturalsFrom (((0 + 1) + 1) + 1)`, protože ho funkce `(!!)` nepotřebuje[1]. Při striktní strategii by jednak vypadalo jinak pořadí vyhodnocování podvýrazů, ale také by se vyhodnocovaly všechny. To by právě u zmíněného podvýrazu vedlo k nekonečnému výpočtu.

Řeš. 5.2.3

a) Funkce `f` při pokusu o vyhodnocení cyklí: `f` $\rightsquigarrow^*$ `f` $\rightsquigarrow^*$ `f` $\rightsquigarrow^*$ ... Avšak opět, funkce `fst` vybere z uvedené dvojice jenom první prvek. Tedy k vyhodnocení `f` nedojde a celý výraz bude vyhodnocen v konečném čase.

b) Funkce `f` je definována jen pro prázdný seznam, ale ve výrazu je volána na neprázdném seznamu. Normálně bychom tedy dostali chybovou zprávu `Non-exhaustive patterns in function f`. Ale protože funkce `const` nepoužívá svůj druhý argument, k vyhodnocení `f [1]` díky lenosti nikdy nedojde, a vyhodnocení tedy skončí bez chyby.

c) Výraz `div 2 0` sám o sobě vrátí chybu `divide by zero`. Může se zdát, že tady zafunguje líné vyhodnocování a `0 * div 2 0` se vyhodnotí na `0`, protože první argument je `0`. Obecně v tomto případě to však není pravda, protože u aritmetických operátorů vždy dochází k vyhodnocení obou operandů. Kvůli efektivitě totiž není vyhodnocování aritmetických operací definováno přímo v Haskellu, ale pomocí primitivních operací procesoru.

d) Při pokusu o vyhodnocení tohoto výrazu dostaneme typovou chybu. Je potřeba mít na paměti, že syntaktická a typová analýza výrazu předchází jeho vyhodnocování, a tedy líné vyhodnocování situaci nezachrání, protože k němu vůbec nedojde.

Řeš. 5.2.4

```
addNumbers :: [String] -> [String]
addNumbers = zipWith (\i n -> show i ++ ". " ++ n) [1 ..]
```

Řeš. 5.2.5

```
integers = 0 : [ sgn * n | n <- [1 ..], sgn <- [1, -1] ]
```

Při tvorbě nekonečných seznamů si musíme dát pozor na to, aby byl každý prvek dosažitelný na konečné pozici. Například řešení `[0 ..] ++ [-1, -2 ..]` toto nesplňuje – všechna záporná čísla se nachází až za nekonečným počtem kladných. V intensionálních seznamech tento problém nastává, pokud se nekonečný generátor objeví na jiné než první pozici. Uvažte třeba nesprávné řešení: `[ sgn * n | sgn <- [1, -1], n <- [1 ..] ]`. Při vyhodnocování se nejdříve zafixuje hodnota `sgn = 1` a následně probíhá nekonečně mnoho dosazení do `n`, takže na volbu `sgn = -1` nikdy nedojde.

[1] Přesněji řečeno, formální parametr `xs`, na který se výraz naváže, se nevyskytuje na pravé straně použité definice funkce `(!!)`.

Řeš. 5.2.6
```
threeSum = [ (x, y, z) | z <- [2 ..], x <- [1 .. z - 1],
                         let y = z - x ]
```

Jak bylo nastíněno v řešení 5.2.5, nekonečný generátor se nesmí objevit na jiné než první pozici, natož aby jich nekonečných bylo více. Nemůžeme tedy napsat třeba `[ (x, y, z) | x <- [1 ..], y <- [1 ..], let z = x + y ]`, protože bychom dostali jen nekonečně mnoho trojic tvaru `(1, y, 1 + y)`. Je proto zapotřebí jít do nekonečna po nějaké vhodné vlastnosti, kterou má vždy jen konečně mnoho prvků. Zde se přímo nabízí součet prvních dvou prvků – pro jeden konkrétní součet snadno vygenerujeme všechny vyhovující trojice, kterých je konečný počet.

Řeš. 5.2.7 Označme počty kroků při líném, normálním a striktním vyhodnocování popořadě L, N a S.

a) $L \leq S$. Nejdříve vyšetříme nekonečné výpočty:
- $L = \infty$, tedy líná strategie vede k zacyklení. Potom dle věty o perpetualitě také striktní strategie vede k zacyklení a $L = S = \infty$.
- $L \neq \infty$ a $S = \infty$, zřejmě $L \leq S$.
- $L \neq \infty$ a $S \neq \infty$. Striktní strategie vynutí vyhodnocení každého podvýrazu právě jednou, kdežto líná nejvýše jednou, obě přitom bez ohledu na počet použití výsledku výrazu (Např. u `f x = x + x` bude výraz dosazený za `x` vyhodnocen pouze jednou). Opět tedy $L \leq S$.

b) Mezi N a S obecně vztah není. První dva případy z předchozí argumentace projdou stejně; ukážeme však, že může nastat $N > S$. Použijeme opět funkci `f x = x + x`. Odkrokujme si vyhodnocení výrazu `f (1 + 2)` oběma strategiemi:
- Striktní: `f (1 + 2)` ⇝ `f 3` ⇝ `3 + 3` ⇝ `6`
- Norm.: `f (1 + 2)` ⇝ `(1 + 2) + (1 + 2)` ⇝ `3 + (1 + 2)` ⇝ `3 + 3` ⇝ `6`

Řeš. 5.2.9

a)
```
repeat True
cycle [True]
iterate id True
```

b)
```
iterate (2 *) 1
```

c)
```
iterate (9 *) 1
```

d)
```
iterate (9 *) 3
```

e)
```
iterate ((-1) *) 1
iterate negate 1
cycle [1, -1]
```

f)
```
iterate ('*' :) ""
iterate ("*" ++) ""
```

g)
```
iterate (\x -> (mod (x + 1) 4)) 1
cycle [1, 2, 3, 0]
```

Řeš. 5.2.10 Chceme-li něco provést pro každé dva sousední prvky seznamu, použijeme „zipování s vlastním ocasem“:

```
differences :: [Integer] -> [Integer]
differences xs = zipWith (-) (tail xs) xs
```

Řeš. 5.2.11
```
values :: (Integer -> a) -> [a]
values f = map f naturals
```

a) První derivaci (diskrétní) funkce `f`.

b) Druhé derivaci (diskrétní) funkce `f`.

Nenecháme se zmást druhou derivací. Tu můžeme použít k vyšetření extrémů jen

u spojitých funkcí. První derivace nám ale pomůže; hledáme body, do nichž funkce klesá (a tedy první derivace je záporná) a z nichž roste (a tedy první derivace následujícího bodu je kladná).

```
localMinima :: (Integer -> Integer) -> [Integer]
localMinima f = map fst3 . filter lmin $ zip3 (tail fs) dfs (tail dfs)
    where
        fs  = values f
        dfs = differences fs
        lmin (_, din, dout) = din < 0 && dout > 0
        fst3 (x, _, _) = x
```

Řeš. 5.2.12 Existuje více řešení. Označíme je postupně `fib`*N*.

```
-- standardni, ale neefektivni definice
fib1 :: Integer -> Integer
fib1 0 = 0
fib1 1 = 1
fib1 n = fib1 (n - 1) + fib1 (n - 2)

-- kompaktnejsi zapis fib1
fib2 :: Integer -> Integer
fib2 n = if n == 0 || n == 1 then n else fib2 (n - 1) + fib2 (n - 2)

-- efektivni seznamova definice
fib3 :: [Integer]
fib3 = fib' (0, 1)
    where
        fib' (x, y) = x : fib' (y, x + y)

-- efektivni definice funkce s akumulacnim parametrem, odvozena z fib3
fib4 :: Integer -> Integer
fib4 n0 = fib' n0 (0, 1)
    where
        fib' 0 (x, y) = x
        fib' n (x, y) = fib' (n - 1) (y, x + y)
```

Různá další řešení lze nalézt na stránce http://www.haskell.org/haskellwiki/The_Fibonacci_sequence.

Řeš. 5.2.13

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Řeš. 5.2.14

```
treeTrim :: BinTree a -> Integer -> BinTree a
treeTrim Empty _ = Empty
treeTrim (Node v l r) 0 = Node v Empty Empty
treeTrim (Node v l r) n =
    Node v (treeTrim l (n-1)) (treeTrim r (n-1))
```

a)
```
treeRepeat :: a -> BinTree a
treeRepeat x = Node x (treeRepeat x) (treeRepeat x)
```

b)
```
treeIterate :: (a -> a) -> (a -> a) -> a -> BinTree a
treeIterate f g x =
```

```
        Node x (treeIterate f g (f x)) (treeIterate f g (g x))
```

c)
```
depthTree :: BinTree Integer
depthTree = treeIterate (+1) (+1) 0
```

Řeš. 5.2.15
```
infFinTree = Node "strom" infFinTree (Node "konec" Empty Empty)
```

Řeš. 5.2.16
```
nonNegativePairs = [ (x, y) | z <- [2 ..], x <- [1 .. z - 1],
                              let y = z - x ]
```

Řešení je prakticky totožné jako 5.2.6, jen se přes součet iteruje „skrytě“.

Řeš. 5.2.17 Je potřeba zvolit vhodnou vlastnost, přes niž se dá iterovat do nekonečna a má ji vždy konečný počet prvků. Vhodnou touto je díky kladnosti prvků součet seznamu. Všechny seznamy s daným součtem vyrobíme pomocí rekurze a intensionálního seznamu.

```
positiveLists = [ l | s <- [0 ..], l <- listsOfSum s ]
    where
        listsOfSum :: Integer -> [[Integer]]
        listsOfSum 0 = [[]]
        listsOfSum n = [x : l | x <- [1 .. n], l <- listsOfSum (n - x)]
```

Řeš. 5.3.1 a)
```
product' :: Num a => [a] -> a
product' []      = 1
product' (x : s) = x * product' s
```

b)
```
length' :: [a] -> Int
length' []      = 0
length' (_ : s) = 1 + length' s
```

c)
```
map' :: (a -> b) -> [a] -> [b]
map' _ []      = []
map' f (x : s) = f x : map' f s
```

Vždy jde o definici, která vrací určitou hodnotu na prázdném seznamu. V případě neprázdného seznamu se výsledek nějakým způsobem získá z prvního prvku seznamu a výsledku rekurzivního volání definované funkce na zbytku seznamu.

Funkcionalitu těchto tří funkcí lze tedy abstrahovat na funkci, která dostane jako jeden argument hodnotu vracenou na prázdném seznamu a jako druhý argument funkci, která se aplikuje na první prvek neprázdného seznamu a na výsledek volání požadované funkce na zbytku seznamu. Tedy:

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' _ z []      = z
foldr' f z (x : s) = f x (foldr' f z s)
```

Řeš. 5.3.2 Pro některé podúlohy je uvedeno více ekvivalentních řešení – tato jsou uváděna pod sebou a jsou odlišena apostrofem v názvu funkce.

a)
```
sumFold :: Num a => [a] -> a
sumFold = foldr (+) 0

sumFold' :: Num a => [a] -> a
sumFold' = foldr (\e t -> e + t) 0
```

b)
```
productFold :: Num a => [a] -> a
productFold = foldr (*) 1
```

c)
```
orFold :: [Bool] -> Bool
orFold = foldr (||) False
```

d)
```
lengthFold :: [a] -> Int
lengthFold = foldr (\_ t -> t + 1) 0
```

e)
```
maximumFold :: Ord a => [a] -> a
maximumFold = foldr1 max

maximumFold' :: Ord a => [a] -> a
maximumFold' list = foldr max (head list) list
```

Řeš. 5.3.3 Jedná se o funkce `sum`, `product`, `or`, `and`, `length`, `minimum` a `maximum`.

Řeš. 5.3.4

a)
```
foldr (-) 0 [1, 2, 3] ⇝* 1 - (2 - (3 - 0))
      ⇝* 2
```

b)
```
foldl (-) 0 [1, 2, 3] ⇝* ((0 - 1) - 2) - 3
      ⇝* -6
```

c)
```
foldr (&&) True (True : False : repeat True)
  ⇝ True && foldr (&&) True (False : repeat True)   {- z def. foldr -}
  ⇝ foldr (&&) True (False : repeat True)           {- z def. (&&) -}
  ⇝ False && foldr (&&) True (repeat True)          {- z def. foldr -}
  ⇝ False                                           {- z def. (&&) -}
```

d)
```
foldl (&&) True (True : False : repeat True)
  ⇝ foldl (&&) (True && True)
               (False : repeat True)                {- z def. foldl -}
  ⇝ foldl (&&) ((True && True) && False)
               (repeat True)                        {- z def. foldl -}
  ⇝ foldl (&&) ((True && True) && False)
               (True : repeat True)                 {- z def. repeat -}
  ⇝ foldl (&&) (((True && True) && False) && True)
               (repeat True)                        {- z def. foldl -}
  ⇝ foldl (&&) (((True && True) && False) && True)
               (True : repeat True)                 {- z def. repeat -}
  ⇝ {- … neskončí -}
```

Všimněte si, že u `foldl` můžeme hodnotu vrátit jen v okamžiku, kdy dojdeme na prázdný seznam. Naopak chování `foldr` na nekonečném seznamu závisí na lenosti dodané funkce.

Řeš. 5.3.5 Jasným kandidátem na řešení je použití některé z akumulačních funkcí. Celkem máme na výběr ze čtyř: `foldr`, `foldl`, `foldr1`, `foldl1`. Připomeňme si, jak která z nich funguje:

```
foldr  (@) w [1,2,3,4] = 1 @ (2 @ (3 @ (4 @ w)))
foldr1 (@)   [1,2,3,4] = 1 @ (2 @ (3 @ 4))
foldl  (@) w [1,2,3,4] = (((w @ 1) @ 2) @ 3) @ 4
foldl1 (@)   [1,2,3,4] = ((1 @ 2) @ 3) @ 4
```

Na základě těchto příkladů vidíme, že jediným vhodným kandidátem na přirozenou definici funkce `subtractlist` je `foldl1`. Tedy ve výsledku dostaneme:

```
subtractlist :: [Integer] -> Integer
subtractlist = foldl1 (-)
```

Řeš. 5.3.6 a) Funkce `foldr` pracuje na seznamech a nahrazuje `(:)` za funkci, v tomto případě za `(.)`, a `[]` za `id`. Intuitivně musí jít o seznam funkcí, které budeme postupně skládat. Ve výsledku tedy vytvoříme složení funkcí v pořadí, v jakém jsou uvedeny v seznamu.

b) Zkusme nejprve otypovat funkci intuitivně. Pro zkrácení řekněme `f = foldr (.) id`.

- Z použití `foldr` plyne, že funkce bude očekávat jako 1. argument seznam.
- Pokud tento seznam bude prázdný, pak `foldr` vrátí `id`, dostáváme tedy zatím typ `[a] -> b -> b` (tento typ je zatím nejspíš příliš obecný, protože jsme nevzali v úvahu všechno chování `foldr`).
- Funkce, která je argumentem `foldr` dostává vždy jako první argument prvek ze seznamu a jako druhý argument výsledek rekurzivního zpracování volání `foldr` na celém zbytku seznamu (což je pro poslední prvek seznamu koncová hodnota, zde `id`). Tím pádem v seznamu musí být funkce, které půjde složit s funkcí `id`.
- Zároveň, tím, že funkce `f` všechny tyto funkce složí za sebe, tak je jich výsledek musí být stejného typu, jako jejich vstup – tak, aby na sebe mohli navázat.
- Dostáváme tedy, že vstupní seznam bude typu `[a -> a]`.
- Složení takových funkcí je pak také typu `a -> a`, což rovněž odpovídá typu `id`.
- Celkem tedy dostáváme `f :: [a -> a] -> a -> a`.

Alternativně můžeme funkci otypovat algoritmicky.

- Máme daný výraz `foldr (.) id`
- Zjistíme si typy všech funkcí:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(.)   :: (a -> b) -> (c -> a) -> c -> b
id    :: a -> a
```

- Přejmenujeme typové proměnné, aby měl každý výskyt každé funkce vlastní typové proměnné:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
(.)   :: (d -> e) -> (f -> d) -> f -> e
id    :: c -> c
```

- Určíme typové rovnosti na základě aplikací:

```
(d -> e) -> (f -> d) -> f -> e = a -> b -> b
c -> c = b
```

- Rozepíšeme typové rovnosti do jednodušších:

```
d -> e = a
f -> d = b
f -> e = b
c -> c = b
```

- Vyjádříme si všechny proměnné pomocí co nejmenšího počtu proměnných:

```
d = e = f = c
b = c -> c
a = c -> c
```

- Zjistíme, jaký typ vlastně hledáme. Je to typový výraz odpovídající typu `foldr` s odstraněnými dvěma typovými argumenty, tedy ve výsledku `[a] -> b`. Dosadíme do něj vyjádření proměnných získaných v předchozím kroku a tím dostaneme výsledný typ:

```
foldr (.) id :: [a] -> b = [c -> c] -> c -> c
```

c) `foldr (.) id [(+ 4), (* 10), (42 ^)]`
d) `foldr (.) id [(+ 4), (* 10), (42 ^)] 1`

Řeš. 5.3.7
```
append' :: [a] -> [a] -> [a]
append' xs ys = foldr (:) ys xs
append'' :: [a] -> [a] -> [a]
append'' = flip (foldr (:))
```

Řeš. 5.3.8
```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

Řeš. 5.3.9 Pro některé podúlohy je uvedeno více ekvivalentních řešení – tato jsou uváděna pod sebou a jsou odlišena apostrofem v názvu funkce.

a)
```
concatFold :: [[a]] -> [a]
concatFold = foldr (++) []
```

b)
```
listifyFold :: [a] -> [[a]]
listifyFold = foldr (\x s -> [x] : s) []
listifyFold':: [a] -> [[a]]
listifyFold' = foldr ((:) . (: [])) []
```

c)
```
nullFold :: [a] -> Bool
nullFold = foldr (\_ _ -> False) True
```

d)
```
composeFold :: [(a -> a)] -> a -> a
composeFold = foldr (.) id
composeFold' :: [(a -> a)] -> a -> a
composeFold' = flip (foldr id)
```

e)
```
idFold :: [a] -> [a]
idFold = foldr (:) []
idFold' :: [a] -> [a]
idFold' = foldr (\e t -> e : t) []
```

f)
```
mapFold :: (a -> b) -> [a] -> [b]
mapFold f = foldr (\e t -> f e : t) []
```

g)
```
headFold :: [a] -> a
headFold = foldr1 const
headFold':: [a] -> a
headFold' = foldr1 (\e t -> e)
```

h)
```
lastFold :: [a] -> a
lastFold = foldr1 (flip const)
lastFold':: [a] -> a
lastFold' = foldr1 (\e t -> t)
```

i)
```
maxminFold :: Ord a => [a] -> (a,a)
maxminFold (x:xs) = foldl (\(ma, mi) e -> (e `max` ma, e `min` mi))
                          (x, x) xs
```

Poznámka: pokud bychom rozbalili definici `foldl`, je toto řešení v podstatě ekvivalentní tomu rekurzivnímu v 5.η.6.

j)
```
suffixFold :: [a] -> [[a]]
suffixFold = foldr (\e (x : xs) -> (e : x) : x : xs) [[]]
```

k)
```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold p = foldr (\e t -> if p e then e : t else t) []
```

l)
```
oddEvenFold :: [a] -> ([a], [a])
oddEvenFold = foldr (\x (l, r) -> (x : r, l)) ([], [])
```

m)
```
takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold p = foldr (\e t -> if p e then e : t else []) []
```

n)
```
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold p = foldl (\t e ->
    if null t && p e
        then []
        else t ++ [e]) []
dropWhileFold':: (a -> Bool) -> [a] -> [a]
dropWhileFold' p list = foldl (\t e ->
    if null (t []) && p e
        then id
        else t . (e :) ) id list []
```

Druhé uvedené řešení má lepší složitost.

Řeš. 5.3.10
```
foldl' f z s = foldr (flip f) z (reverse s)
```

Řeš. 5.3.11
```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x < y
    then x : y : ys
    else y : insert x ys

insert' :: Ord a => a -> [a] -> [a]
insert' x = foldr (\y (z : zs) ->
    if y > z
        then z : y : zs
        else y : z : zs) [x]
```

Poznamenejme, že první varianta je díky lenosti tím rychlejší, čím blíže začátku se má prvek vložit, kdežto druhá varianta vždy prochází a kopíruje celý seznam.

```
insertSort :: Ord a => [a] -> [a]
insertSort = foldr insert []
```

Řeš. 5.3.12 Funkce `foldr` přestane vyhodnocovat výraz po prvním nalezeném `True` (díky línému vyhodnocování funkce `(||)`), avšak `foldl` ho vždy projde celý. Tedy v případě nekonečného seznamu `foldr` skončí po prvním nalezeném `True`, ale `foldl` neskončí nikdy.

Řeš. 5.3.13 Odpověď jste měli hledat na internetu, ne tady.

Řeš. 5.3.14 Ne, není to možné. Funkce `f` by musela dokázat rozlišit, kdy je volána na prvku ze sudého místa a kdy z lichého. K dispozici má však pouze n-tý prvek a seznam vzniklý zpracováním

prvních $n-1$ prvků.

Důkaz. Předpokládejme pro spor, že taková funkce `f` existuje.

Ukážeme nejdřív, že funkce `f` si nemůže „dělat poznámky“ ve svém výsledku na to, aby poznala, zda je na sudé či liché pozici, ale musí ve svém n-tém volání vracet přímo výsledek na prvních n prvcích. Toto tvrzení je přímým důsledkem toho, že potřebujeme po poslední aplikaci funkce `f` dostat rovnou hledaný výsledek (a že v okamžiku kdy aplikaci provádíme, nemáme jak poznat, zda je poslední).

Uvažme dále seznamy `as = [1, 2, 3]` a `bs = [1, 3]`. Pak v nějakém momentě dojde pro oba seznamy k volání `f [1] 3`. V případě seznamu `as` však má dojít k přidání `3` do výsledku, zatímco v případě `bs` ne. To však není možné – funkce `f` je čistá funkce, nemá tedy jak tyto dva případy rozlišit. □

Ve druhém případě to možné je:

```
f = \(b, s) x -> (not b, if b then s ++ [x] else s)
v = (True, [])
```

Teď již postupujeme zleva (`foldl`) a v hodnotě typu `Bool` si ukládáme, jestli je aktuální pozice sudá.

Řeš. 5.3.15

```
foldr f z s = foldr2 (\x y s -> f x (f y s)) (\x -> f x z) z s
```

Opačná definice (`foldr2` pomocí `foldr`) není možná. Pomocí `foldr2` je možné vybrat každý druhý prvek seznamu (`foldr2 (\x y s -> x : s) (: []) []`), což však pomocí `foldr` není možné (viz úloha 5.3.14).

Řeš. 5.4.1

```
treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold _ e Empty        = e
treeFold n e (Node v l r) = n v (treeFold n e l) (treeFold n e r)
```

Řeš. 5.4.2

a)
```
treeSize :: BinTree a -> Int
treeSize = treeFold (\_ l r -> 1 + l + r) 0
```

b)
```
treeHeight :: BinTree a -> Int
treeHeight = treeFold (\_ l r -> 1 + max l r) 0
```

c)
```
treeList :: BinTree a -> [a]
treeList = treeFold (\v l r -> l ++ [v] ++ r) []
```

d)
```
treeConcat :: BinTree [a] -> [a]
treeConcat = treeFold (\v l r -> l ++ v ++ r) []
```

e)
```
treeMax :: (Ord a, Bounded a) => BinTree a -> a
treeMax = treeFold (\v l r -> maximum [v, l, r]) minBound
```

f)
```
treeFlip :: BinTree a -> BinTree a
treeFlip = treeFold (\v l r -> Node v r l) Empty
```

g)
```
treeId :: BinTree a -> BinTree a
treeId = treeFold (\v l r -> Node v l r) Empty
treeId' = treeFold Node Empty
```

h)
```
rightMostBranch :: BinTree a -> [a]
rightMostBranch = treeFold (\v l r -> v:r) []
```

i)
```
treeRoot :: BinTree a -> a
treeRoot = treeFold (\v l r -> v) undefined
treeRoot' = treeFold (const . const) undefined
```

j)
```
treeNull :: BinTree a -> Bool
treeNull = treeFold (\v l r -> False) True
```

k)
```
leavesCount :: BinTree a -> Int
leavesCount = treeFold (\v l r -> if l + r == 0 then 1 else l + r) 0
```

l)
```
leavesList :: BinTree a -> [a]
leavesList = treeFold (\v l r ->
    if null l && null r
        then [v]
        else l ++ r) []
```

m)
```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap f = treeFold (\v l r -> Node (f v) l r) Empty
treeMap' f = treeFold (\v -> Node (f v)) Empty
treeMap'' f = treeFold (Node . f) Empty
```

n)
```
treeAny :: (a -> Bool) -> BinTree a -> Bool
treeAny p = treeFold (\v l r -> p v || l || r) False
treeAny' p = treeFold (\v l r -> or [p v, l, r]) False
```

o)
```
treePair :: Eq a => BinTree (a,a) -> Bool
treePair = treeFold (\(x,y) l r -> x == y && l && r) True
```

p)
```
subtreeSums :: Num a => BinTree a -> BinTree a
subtreeSums = treeFold (\v l r -> Node (v + root l + root r) l r)
                       Empty
    where
        root (Node v l r) = v
        root Empty = 0
```

Řeš. 5.4.3
```
roseTreeFold :: (a -> [b] -> b) -> RoseTree a -> b
roseTreeFold n (RoseNode v ts) = n v (map (roseTreeFold n) ts)
```

Řeš. 5.4.4

a)
```
roseTreeSize :: RoseTree a -> Int
roseTreeSize = roseTreeFold (\_ xs -> 1 + sum xs)
```

b)
```
roseTreeHeight :: RoseTree a -> Int
roseTreeHeight = roseTreeFold (\_ xs -> 1 + maximum (0:xs))
```

Všimněte si přidání 0 proto aby maximum fungovalo i pro listy. To v tomto případě můžeme bezpečně udělat, pokud seznam prázdný nebude, nic to nerozbije. Víme totiž, že výška nemůže být záporná a že seznam je seznamem čísel.

c) Inorder průchod se u těchto stromů nedá dobře definovat, můžeme ale udělat třeba preorder – v seznamu je nejprve hodnota uzlu a následně hodnoty všech podstromů.

```
roseTreeList :: RoseTree a -> [a]
roseTreeList = roseTreeFold (\v xs -> v : concat xs)
```

d)
```
roseTreeConcat :: RoseTree [a] -> [a]
roseTreeConcat = roseTreeFold (\v xs -> v ++ concat xs)
```

e)
```
roseTreeMax :: (Ord a, Bounded a) => RoseTree a -> a
roseTreeMax = roseTreeFold (\v xs -> maximum (v : xs))
```

f)
```
roseTreeFlip :: RoseTree a -> RoseTree a
roseTreeFlip = roseTreeFold (\v xs -> RoseNode v (reverse xs))
```

g)
```
roseTreeId :: RoseTree a -> RoseTree a
roseTreeId = roseTreeFold (\v xs -> RoseNode v xs)
```

```
roseTreeId' = roseTreeFold RoseNode
```

h)
```
roseRightMostBranch :: RoseTree a -> [a]
roseRightMostBranch = roseTreeFold (\v xs -> v :
    if null xs then [] else last xs)
```

i)
```
roseTreeRoot :: RoseTree a -> a
roseTreeRoot = roseTreeFold (\v _ -> v)
roseTreeRoot' = roseTreeFold const
```

j) Datový typ neumožňuje reprezentovat prázdné stromy.

```
roseTreeNull :: RoseTree a -> Bool
roseTreeNull = roseTreeFold (\_ _ -> False)
roseTreeNull' = roseTreeFold (const . const False)
```

k)
```
roseLeavesCount :: RoseTree a -> Int
roseLeavesCount = roseTreeFold (\_ xs ->
    if null xs then 1 else sum xs)
```

l)
```
roseLeavesList :: RoseTree a -> [a]
roseLeavesList = roseTreeFold (\v xs ->
    if null xs then [v] else concat xs)
```

m)
```
roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap f = roseTreeFold (\v xs -> RoseNode (f v) xs)
roseTreeMap' f = roseTreeFold (\v -> RoseNode (f v))
roseTreeMap'' f = roseTreeFold (RoseNode . f)
```

n)
```
roseTreeAny :: (a -> Bool) -> RoseTree a -> Bool
roseTreeAny p = roseTreeFold (\v xs -> p v || or xs)
roseTreeAny' p = roseTreeFold (\v xs -> or (p v : xs))
```

o)
```
roseTreePair :: Eq a => RoseTree (a, a) -> Bool
roseTreePair = roseTreeFold (\(x, y) xs -> x == y && and xs)
roseTreePair' :: Eq a => RoseTree (a, a) -> Bool
roseTreePair' = roseTreeFold (\(x, y) xs -> and ((x == y) : xs))
```

p)
```
roseSubtreeSums :: Num a => RoseTree a -> RoseTree a
roseSubtreeSums = roseTreeFold (\v xs -> RoseNode
                                          (sum (v : map root xs))
                                          xs)
    where root (RoseNode v _) = v
```

Řeš. 5.4.5 Nejprve je třeba promyslet si, jak by taková funkce intuitivně měla fungovat. Katamorfismus je obecně funkce na struktuře, která nahrazuje konstruktory této struktury zadanými funkcemi nebo hodnotami, a ve výsledku umožní rekurzivně projít celou strukturu. Datový typ `Nat` má konstruktory `Zero :: Nat` a `Succ :: Nat -> Nat`. Našim cílem je převod hodnoty tohoto typu na nějakou hodnotu, obecně typu `a`. Katamorfismus na hodnotách daného typu je definován funkcemi, které nahrazují jeho hodnotové konstruktory funkcemi stejné arity, jejichž výsledná hodnota je typu `a`, a v místě, kde má konstruktor argument původního typu (v tomto případě tedy `Nat`), uvedeme `a`.

S těmito znalostmi se tedy podívejme na typ `Nat`. Hodnotový konstruktor `Zero` nahradíme nulární funkcí, tedy hodnotou typu `a`. Hodnotový konstruktor `Succ` nahradíme unární funkcí s typem `a -> a`. Když definujeme tuto transformaci jako funkci, musíme ji definovat po částech pro jednotlivé hodnotové konstruktory. V těle pak použijeme dodané funkce a rekurzivně voláme `natFold`:

```
natFold :: (a -> a) -> a -> Nat -> a
natFold s z Zero     = z
natFold s z (Succ x) = s (natFold s z x)
```

Pokud bychom fixovali parametry `s` a `z`, lze lépe vidět, jak katamorfismus na `Nat` pracuje:

```
natFoldsz :: Nat -> a
natFoldsz Zero     = z
natFoldsz (Succ x) = s (natFoldsz x)
```

Řeš. 5.4.6 a) Číslo n typu `Nat` vlastně znamená přičtení n jedniček k nule. Když nezačneme nulou, ale jiným číslem m, dostaneme přesně součet $n+m$. Budeme tedy foldovat jedno z čísel, konstruktory `Succ` necháme být a `Zero` nahradíme druhým číslem.

```
natAdd :: Nat -> Nat -> Nat
natAdd m n = natFold Succ m n
natAdd' = natFold Succ
```

b) Nula je sudá, konstruktor `Zero` se tedy má vyhodnotit na `True`. Přičtení jedničky vždy paritu změní, konstruktory `Succ` proto nahradíme funkcí invertující logickou hodnotu.

```
natEven :: Nat -> Bool
natEven = natFold not True
```

c) Součin m a n je jen n-násobné přičtení m k nule. Přičítat už umíme funkcí `natAdd` a chceme-li funkci provést n-krát, nahradíme jí všechny konstruktory `Succ` v n.

```
natMul :: Nat -> Nat -> Nat
natMul m n = natFold (natAdd m) Zero n
```

Řeš. 5.4.7 Implementace `Foldable` je možná pomocí funkcí `foldr` nebo `foldMap`. Funkce `foldr` v tomto případě vede na následující, poměrně složitou, definici.

```
instance Foldable BinTree where
    -- type cannot be specified in instance
    -- foldr :: (a -> b -> b) -> b -> BinTree a -> b
    foldr _ e Empty = e
    foldr f e (Node v l r) = f v (foldr f (foldr f e r) l)
```

Zatímco bázový případ je poměrně přímočarý, ten rekurzivní není – v tomto případě máme problém s tím, že nemáme k dispozici nic, čím bychom dokázali zkombinovat více hodnot typu `b`, vyjma funkce `f`, která ale ještě bere argument typu `a`. Musíme tedy porušit symetrii přístupu k levému a pravému podstromu a výsledek zpracování jednoho z nich dát jako bázový případ pro druhý.

Jednodušší alternativou je postup s využitím *monoidů*. *Monoid* je daný nosnou množinou, binární operací a jejím neutrálním prvkem. Je třeba aby množina byla uzavřená vzhledem k dané binární operaci a operace byla asociativní. V kontextu typové třídy `Monoid` je operace reprezentována operátorem `(<>)` a neutrální prvek hodnotou `mempty`.

Instanci `Foldable` můžeme implementovat pomocí funkce `foldMap`, která provede projekci všech prvků stromu do daného monoidu a výsledky pospojuje monoidovou operací `(<>)`.

```
instance Foldable BinTree where
    -- type cannot be specified in instance
    -- foldMap :: Monoid m => (a -> m) -> BinTree a -> m
    foldMap proj tree = go tree
      where
```

```
go Empty = mempty
go (Node val left right) = proj val <> go left <> go right
```

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.

05_data.hs

```haskell
data Sex = Female | Male

allPairs :: [(String, Sex)] -> [(String, String)]
allPairs = undefined

people :: [(String, Sex)]
people =  [("Jeff", Male), ("Britta", Female), ("Annie", Female), ("Troy",
↪  Male)]


naturalsFrom :: Integer -> [Integer]
naturalsFrom n = n : naturalsFrom (n + 1)

naturals :: [Integer]
naturals = naturalsFrom 0

-- naturals !! 2


filter' _ [] = []
filter' p (x : xs) = if p x then x : filter' p xs else filter' p xs

-- Jak se bude chovat interpret jazyka Haskell pro vstup filter' (< 3)
↪  naturals?

takeWhile' _ [] = []
takeWhile' p (x : xs) = if p x then x : takeWhile' p xs else []

-- Jak se bude chovat interpret jazyka Haskell pro vstup takeWhile' (< 3)
↪  naturals?

addNumbers :: [String] -> [String]
addNumbers = undefined

integers :: [Integer]
integers = undefined

threeSum :: [(Integer, Integer, Integer)]
threeSum = undefined


sumFold :: Num a => [a] -> a
sumFold = undefined
```

```haskell
productFold :: Num a => [a] -> a
productFold = undefined

orFold :: [Bool] -> Bool
orFold = undefined

lengthFold :: [a] -> Int
lengthFold = undefined

maximumFold :: Ord a => [a] -> a
maximumFold = undefined


-- foldr1 (\x s -> x + 10 * s)
-- foldl1 (\s x -> 10 * s + x)


concatFold :: [[a]] -> [a]
concatFold = undefined

listifyFold :: [a] -> [[a]]
listifyFold = undefined

nullFold :: [a] -> Bool
nullFold = undefined

composeFold :: [a -> a] -> a -> a
composeFold = undefined

idFold :: [a] -> [a]
idFold = undefined

mapFold :: (a -> b) -> [a] -> [b]
mapFold = undefined

headFold :: [a] -> a
headFold = undefined

lastFold :: [a] -> a
lastFold = undefined

maxminFold :: Ord a => [a] -> (a, a)
maxminFold = undefined

suffixFold :: [a] -> [[a]]
suffixFold = undefined

filterFold :: (a -> Bool) -> [a] -> [a]
filterFold = undefined

oddEvenFold :: [a] -> ([a], [a])
oddEvenFold = undefined

takeWhileFold :: (a -> Bool) -> [a] -> [a]
takeWhileFold = undefined
```

```haskell
dropWhileFold :: (a -> Bool) -> [a] -> [a]
dropWhileFold = undefined
```

05_treeFold.hs

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

treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold n e (Node v l r) = n v (treeFold n e l)
                                (treeFold n e r)
treeFold n e Empty        = e

tree01 :: BinTree Int
tree01 = Node 2 (Node 3 (Node 5 Empty Empty) Empty)
                (Node 4 (Node 1 Empty Empty) (Node 1 Empty Empty))

tree02 :: BinTree String
tree02 = Node "C" (Node "A" Empty (Node "B" Empty Empty))
                  (Node "E" (Node "D" Empty Empty) Empty)

tree03 :: BinTree (Int,Int)
tree03 = Node (3,3) (Node (2,1) Empty Empty) (Node (1,1) Empty Empty)

tree04 :: BinTree a
tree04 = Empty

tree05 :: BinTree Bool
tree05 = Node False (Node False Empty (Node True Empty Empty))
                    (Node False Empty Empty)

tree06 :: BinTree (Int, Int -> Bool)
tree06 = Node (0,even)
              (Node (1,odd) (Node (2,(== 1)) Empty Empty) Empty)
              (Node (3,(< 5)) Empty
                              (Node (4,((== 0) . mod 12)) Empty Empty))
```