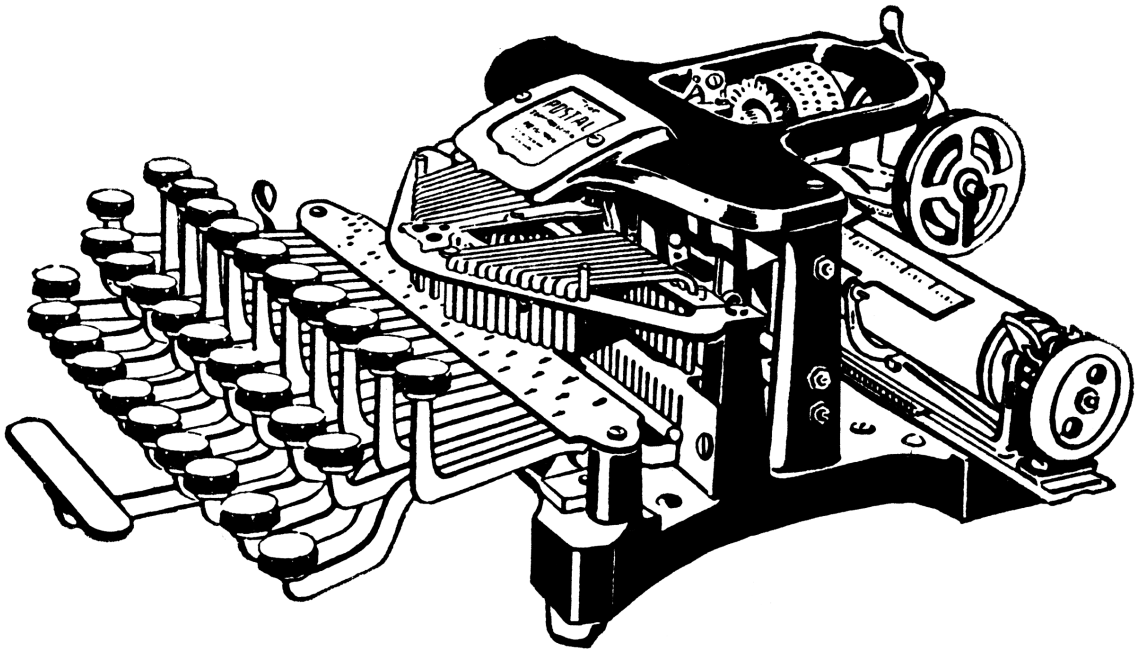
IB015 – Neimperativní programování



Funkcko

Sbírka příkladů ke cvičením

Fakulta informatiky
Masarykovy univerzity

Změněno: 23. 12. 2019 13.58

Obsah

Tato sbírka vzniká přispěním mnoha autorů již několik let a stále se vyvíjí. Narazíte-li na nějakou chybu či nejasnost, nahlaste nám ji, prosíme, do diskusního fóra předmětu.

Podoba úvodní strany je s úctou obšlehnuta od O'Reillyho nakladatelství. Ilustraci psacího stroje poskytlo Florida Center for Instructional Technology: `https://etc.usf.edu/clipart`.

Cvičení 0: Technické okénko

Fialové rámečky na začátku každého cvičení obsahují věci, které je bezpodmínečně nutné znát ještě před začátkem cvičení. Neslouží *jako opakování přednášky, avšak jsou vám ku pomoci při přípravě na cvičení.*

Před nultým, technickým cvičením je zapotřebí znát:

- svoje učo (universitní číslo osoby, správnost si ověříte přihlášením do ISu);
- svoje fakultní přihlašovací jméno (tzv. *xlogin*) a fakultní heslo (to je jiné než do ISu). Svůj *xlogin* zjistíte a heslo můžete změnit na https://is.muni.cz/auth/system/heslo_fi.

Oranžové rámečky obsahují upozornění a varování.

Jindřiška varuje: Na cvičení je normální se připravovat. Nebudete-li s pojmy ve fialových rámečcích srozuměni, pravděpodobně si ze cvičení kromě pocitu neúspěchu příliš neodnesete a hrozí vám vyloučení ze cvičení.

Nulté cvičení je velmi nestandardní. Trvá necelou hodinu a nezabývá se náplní předmětu, ale má za úkol vás seznámit s počítačovou učebnou B130 a jejím programovým vybavením, aby se na prvním cvičení nemusely řešit problémy technického rázu.

Vysvětlivky Při té příležitosti se hodí vysvětlit význam piktogramů objevujících se u některých příkladů v této sbírce:

Symbolem »= jsou označeny ty příklady, které by se měly stihnout na cvičení. Příklady bez tohoto symbolu tak slouží spíše pro domácí studium a přípravu. Mimochodem, přesný význam tohoto symbolu vám bude objasněn na konci semestru; pokud mu chcete nějak říkat, můžete používat termín *bind*. Ještě více mimochodem, podobnost s logem Haskellu není náhodná.

Tužkou () jsou označeny příklady, které byste měli být schopni vyřešit správně s použitím pouze tužky a papíru, stejně jako se to po vás bude chtít u zkoušky.[1] Interpret doporučujeme použít až ke kontrole.

Démantem () označujeme příklady, které sice nejsou v plánu cvičení, ale přesto je velmi doporučujeme vyřešit, neboť je považujeme za zajímavé či zvláště přínosné. Očekáváme, že tyto příklady budete řešit pravidelně v průběhu semestru a mohou vám pomoci při řešení odpovídajících domácích úkolů.

Mírně obtížnější, ale o to poutavější příklady pak nesou hvězdičku () či několik hvězdiček podle obtížnosti či pracnosti. Trojhvězdičkové příklady mohou sahat i za rámec tohoto předmětu.

Zelené rámečky obsahují rady, doporučení a tipy na další zdroje ke studiu.

Pan Fešák doporučuje: K většině příkladů se na konci sbírky nachází řešení. Podporuje-li váš prohlížeč dokumentů odkazy, můžete klepnutím na číslo příkladu na jeho řešení skočit.

Př. 0.0.0 Zkuste si to!

[1] Nepodlehněte však iluzi, že před zkouškou stačí projít tužkou označené příklady a zelenou známku máte v kapse. Piktogram totiž nijak nesouvisí s typem příkladů objevujících se u zkoušky.

0.1 Základní využívání školních počítačů

Př. 0.1.1 Přihlaste se na školní počítač (pomocí xloginu a fakultního hesla). Po přihlášení spusťte terminál (buď jej vyhledejte, nebo pomocí klávesové zkratky Ctrl+Alt+T). Po spuštění by měl terminál pracovat ve vašem domovském adresáři (`/home/xLOGIN`). Příkazem `pwd` si vypište aktuální adresář.

Př. 0.1.2 Vytvořte si ve svém domovském adresáři adresář `ib015`. Můžete využít terminál nebo grafický správce souborů.

Př. 0.1.3 V adresáři `ib015` vytvořete nový soubor `Sem0.hs` s následujícím obsahem:

```
hello = "Hello, world"
```

0.2 Používání GHCi

Př. 0.2.1 V terminálu v adresáři `ib015` si příkazem `ghci` spusťte interpret Haskellu.

Př. 0.2.2 V interpretu si zkuste vyhodnotit jednoduché aritmetické výrazy, tedy využít ho jako kalkulačku.

Př. 0.2.3 Do GHCi načtěte soubor `Sem0.hs` a nechte si vypsat konstantu `hello`. Následně si od interpretu vyžádejte její typ.

Př. 0.2.4 Do souboru `Sem0.hs` přidejte konstantu `my_number` typu `Integer` a nastavte ji na svou oblíbenou hodnotu. Po uložení souboru jej znovu načtěte do GHCi a konstantu vypište. Interpret následně ukončete.

0.3 Vzdálené připojení na fakultní počítače

> **Pan Fešák doporučuje:** Používání vzdáleného připojení přes SSH není v tomto předmětu nezbytně nutné, ale může se vám hodit a určitě jej budete potřebovat jinde, tak proč se to nenaučit již nyní. Vzdálené připojení vám umožní používat GHCi a další nástroje na počítačích na FI a dostat se k vašim kódům ze cvičení odkudkoli.

Na školní stroje se systémem Linux se lze připojit pomocí SSH. Na Linuxu či macOS se obvykle jako klient SSH používá příkaz `ssh`. Pokud klient nemáte, mělo by být možné jej nainstalovat z balíčků (bude se nejspíš jmenovat `openssh-client` nebo `openssh`). Na Windows 10 je příkaz `ssh` k dispozici v PowerShellu. Záložní možností je pak klient Putty.

K přihlašování z vnějšku fakulty slouží server `aisa.fi.muni.cz` (ze sítě FI dostupný i jako `aisa`). Přes tento počítač se pak případně dá dostat i k jiným, ale to nebývá potřeba, protože váš domovský adresář je sdílen přes všechny fakultní linuxové stroje.

Př. 0.3.1 Přihlaste se ze svého nebo školního počítače k serveru `aisa.fi.muni.cz`. Tam přidejte modul s novým GHC pomocí příkazu `module add ghc`[2] a následně načtěte do GHCi kód ze cvičení.

[2] Na Aise je nainstalované příliš staré GHC, které se od toho námi používaného liší typy některých seznamových funkcí.

Př. 0.3.2 
Pomocí příkazu `scp` si na svůj počítač zkopírujte z Aisy zdrojový kód ze cvičení. U sebe jej upravte a pošlete zpět na Aisu. Alternativně můžete vyzkoušet některý z klikacích nástrojů: na Windows WinSCP, v Linuxových grafických správcích souborů hledejte volbu „Připojit k serveru“ nebo podobnou.

Př. 0.3.3 
Nastavte si klient SSH tak, aby pro připojení k Aise stačilo zadat `ssh aisa`. Stejnou zkratku je pak možné využívat při kopírování souborů mezi počítači: `scp Sem0.hs aisa:ib015/`.

> **Pan Fešák doporučuje:** Nevíte-li si kdy v terminálu rady, zkuste před návštěvou svého oblíbeného vyhledávače použít příkaz `man`. Zde se bude hodit `man ssh_config`.

Př. 0.3.4 

> Tento krok je určen pokročilejším uživatelům a sahá daleko nad rámec předmětu.

Pokud nedisponujete dvojicí klíčů SSH pro asymetrické šifrování, vytvořte si ji. Přidejte na Aise svůj veřejný klíč mezi autorizované, abyste při přihlašování ze svého počítače nemuseli psát heslo.

0.4 Nastavení vlastního počítače

> **Jindřiška silně doporučuje** rozchodit si GHC na svém vlastním počítači. Bude se vám hodit při dělání domácích úkolů a přípravě na cvičení.

Př. 0.4.1 
Podle instrukcí v interaktivní osnově předmětu si nainstalujte GHC na svůj počítač a ověřte si, že funguje v rozsahu práce z tohoto cvičení.

*
**

> **Jindřiška varuje:** Ať už budete používat vlastní či školní počítač, je naprosto nezbytné, abyste před prvním cvičením zvládli jeho obsluhu alespoň v rozsahu fialového rámečku na začátku následující kapitoly.

> *Fialové rámečky na konci cvičení shrnují probrané koncepty. Můžete si s jejich využitím ověřit, jestli všemu ze cvičení rozumíte, nebo je potřeba se k nějakému tématu vrátit.*
>
> **Na konci cvičení byste měli zvládnout:**
>
> - přihlásit se k fakultním počítačům v učebně B130;
> - vytvořit textový soubor s Haskellovým kódem;
> - spustit v terminálu GHCi a načíst do něj vytvořený soubor;
> - pracovat v terminálu vzdáleně na školním počítači Aisa.

Cvičení 1: Základní konstrukce

Před cvičením je nezbytné umět:

- přihlásit se k počítačům v učebně B130;
- otevřít terminál (příkazovou řádku) a vytvořit textový soubor;
- spustit interpret GHCi a načíst do něj soubor;
- používat GHCi jako kalkulačku;
- znát na intuitivní úrovni pojmy *funkce* a *typ*;
- vědět, jaké základní typy v Haskellu jsou;
- vědět, jak fungují základní konstrukce `if`, `let ... in ...` a `where`.

Příkazy interpretu GHCi:

`:t[ype]` *`výraz`* – typ výrazu

`:i[nfo]` *`jméno`* – informace o operátorech, funkcích a typech

`:doc` *`jméno`* – dokumentace operátorů, funkcí a typů (až od GHC 8.6)

`:l[oad]` *`soubor.hs`* – načtení souboru s Haskellovým kódem

`:r[eload]` – znovunačtení posledního souboru

`:q[uit]` – ukončení práce s interpretem

`:m[odule]` *`Modul`* – načtení modulu (bude používáno později)

`:h[elp]` – nápověda

1.1 Práce s dokumentací

Tak jako u ostatních programovacích jazyků je většina funkcí a typů zdokumentovaná. Primárním zdrojem dokumentace pro jazyk Haskell v rozsahu našeho kurzu je webová dokumentace základního modulu `Prelude` na **Hackage**. Dále můžete využít vyhledávač **Hoogle**, kde můžete vyhledávat funkce podle názvu nebo typu (*pozor, vyhledává i v nestandardních balíčcích, pokud si nenastavíte* `package:base`).

Jindřiška varuje: Naučit se pracovat s dokumentací je nevyhnutelné pro libovolný programovací jazyk, nejenom pro Haskell. Čím dříve se naučíte číst dokumentaci, tím budete mít lehčí život nejen tu, ale i v dalších předmětech.

Př. 1.1.1 »= Pomocí vyhledávače funkcí v jazyce Haskell najděte všechny funkce, které mají typ `Bool -> Bool -> Bool` a jsou v balíčku `base`.

1.2 Priority operátorů

Př. 1.2.1 S využitím interního příkazu `:info` interpretu GHCi zjistěte prioritu a asociativitu následujících operací:

```
^, *, /, +, -, ==, /=, >, <, >=, <=, &&, ||
```

> **Pan Fešák doporučuje:** Priorita operátorů je jednou z věcí, které jsou nutné pro správné pochopení vyhodnocování výrazů. Je vhodné naučit se používat dotaz `:i`, který mimo jiné obsahuje i prioritu zadaného operátoru.
>
> Příklad dotazu v interpretu (řádek začínající `>` je zadán uživatelem):
>
> ```
> > :i *
> class Num a where
>   ...
>   (*) :: a -> a -> a
>   ...
>         -- Defined in ‘GHC.Num’
> infixl 7 *
> ```
>
> Pro nás je nyní podstatný poslední řádek, který definuje, že se jedná o infixový operátor a popisuje jeho prioritu a asociativitu. V tomto případě `infixl 7 *` znamená, že se jedná o operátor priority 7, který se závorkuje (asociuje) zleva (`infixl`; zprava by bylo `infixr`), tedy například `2 * 3 * 7` se vyhodnocuje jako by bylo uzávorkované `(2 * 3) * 7`. Například pro `==` dostaneme `infix 4 ==`, což znamená, že `==` nelze řetězit s dalšími operátory na stejné úrovni priority a jeho priorita je 4.
>
> Pokud nemá operátor (funkce) explicitně definovanou prioritu, jeho priorita je 9 a je asociativní zleva. I některé binární funkce zapsané písmeny mají určenou prioritu a asociativitu, kterou využijí, pokud jsou zapsány infixově. Příkladem takové funkce je `div`.
>
> Konečně prefixová aplikace má vždy přednost před aplikací infixovou, například ve výrazu `div 3 2 ^ 4` se provede nejprve dělení a až pak umocňování, jako by byl výraz uzávorkovaný `(div 3 2) ^ 4`.

Př. 1.2.2 S použitím interpretu jazyka Haskell porovnejte vyhodnocení následujících dvojic výrazů a rozdíl vysvětlete.

a) `5 + 9 * 3` versus `(5 + 9) * 3`
b) `2 ^ 2 ^ 2 == (2 ^ 2) ^ 2` versus `3 ^ 3 ^ 3 == (3 ^ 3) ^ 3`
c) `3 + 3 + 3` versus `3 == 3 == 3`
d) `("Haskell" == "je") == "super"` versus `('a' == 'a') == ('a' == 'a')`

> **Pan Fešák doporučuje:** Operátory a funkce se můžou vyskytovat v prefixové i infixové verzi. Proto si dohledejte, jaký je rozdíl mezi `*` vs. `(*)` a `mod` vs. `` `mod` ``.

Př. 1.2.3 Přepište infixové zápisy výrazů do syntakticky správných prefixově zapsaných výrazů a naopak:

a) `4 ^ (7 `mod` 5)`
b) `max 3 ((+) 2 3)`

Př. 1.2.4 Doplňte všechny implicitní závorky do následujících výrazů:

a) `recip 2 * 5`
b) `sin pi / 2`

c) `mod 3 8 * 2`
d) `f g 3 + g 5`
e) `42 < 69 || 5 == 6`
f) `2 + div m 18 == m ^ 2 ^ n && m * n < 20`

> **Pan Fešák doporučuje:** Pokud nevíte, co daná funkce nebo operátor dělá, je vhodné si ji najít v dokumentaci. Příkladem může být operátor `(||)`.

Př. 1.2.5 Doplňte všechny implicitní závorky do následujících výrazů:

a) `f . g x`
b) `2 ^ mod 9 5`
c) `f . (.) g h . id`
d) `2 + div m 18 * m `mod` 7 == m ^ 2 ^ n - m + 11 && m * n < 20`
e) `f 1 2 g + (+) 3 `const` g f 10`
f) `replicate 8 x ++ filter even (enumFromTo 1 (3 + 9 `mod` x))`
g) `id id . flip const const`

*
**

> **Pan Fešák doporučuje:** Pokud jste si ještě nevytvořili soubor pro toto cvičení, teď je dobrý čas jej vytvořit. Funkce z dalších příkladů pište do souboru a pak je testujte v GHCi.

Př. 1.2.6 Definujte funkci `isSucc :: Integer -> Integer -> Bool`, která pro dvě celá čísla `x`, `y` rozhodne, jestli je `y` následníkem `x`.

Př. 1.2.7 Vytvořte funkci `circleArea :: Double -> Double`, která pro zadaný poloměr spočítá obsah kruhu o tomto poloměru. Přibližná hodnota konstanty π se dá v Haskellu získat pomocí konstanty `pi`.

Př. 1.2.8 Definujte funkci `max3 :: Integer -> Integer -> Integer -> Integer`, která pro tři celá čísla `x`, `y` a `z` vrátí to největší z nich. Naprogramujte dvě verze, jednu pomocí funkce `max` a jednu pomocí `if ... then ... else ...`.

Př. 1.2.9 Pro zadané tři délky stran trojúhelníka rozhodněte, zda se jedná o pravoúhlý trojúhelník. Pravoúhlý trojúhelník je možné poznat tak, že pro délky jeho stran platí Pythagorova věta (tedy součet druhých mocnin dvou kratších stran je roven druhé mocnině nejdelší strany). V řešení zkuste vhodně využít lokální definici (`where` nebo `let ... in ...`).

```
isRightTriangle 3 4 5    ⇝* True
isRightTriangle 42 42 42 ⇝* False
isRightTriangle 70 42 56 ⇝* True
isRightTriangle 25 24 7  ⇝* True
```

Př. 1.2.10 Naprogramujte funkci `mid :: Integer -> Integer -> Integer -> Integer`, která pro tři celá čísla `x`, `y` a `z` vrátí to *prostřední* z nich (tj. to druhé v jejich uspořádané trojici podle $\leq$).

```
mid 1 2 3 ⇝* 2
mid 42 16 69 ⇝* 42
mid 15 113 111 ⇝* 111
mid 42 42 42 ⇝* 42
```

Př. 1.2.11 
Pomocí `if` a funkce `mod` definujte funkci `tell :: Integer -> String`, která bere jako argument jedno nezáporné celé číslo `n` a vrací:

a) `"one"` pro `n = 1`
b) `"two"` pro `n = 2`
c) `"(even)"` pro sudé `n > 2`
d) `"(odd)"` pro liché `n > 2`

> **Jindřiška varuje:** V porovnání s jinými (převážně imperativními) jazyky má Haskell striktnější pravidla pro konstrukci `if ... then ... else ...` a je nutné je všechna znát.
>
> V imperativních jazycích totiž Haskellovému výrazu `if p then t else f` neodpovídá řídicí příkaz `if p then t else f`, ale ternární operátor: `p ? t : f` (nejen) v jazyce C nebo `t if p else f` v Pythonu.

Př. 1.2.12 
U následujících výrazů rozhodněte, zda jsou správně, a pokud jsou špatně, zdůvodněte proč a vhodným způsobem je upravte.

a) `if 5 - 4 then False else True`
b) `if 0 < 3 && odd 6 then 0 else "FAIL"`
c) `(if even 8 then (&&)) (0 > 7) True`
d) `if 42 < 42 then (&&) else (||)`

> **Pan Fešák doporučuje:** Pokud v konstrukci `if ... then ... else ...` mají větve `then` a `else` typ `Bool`, pak je vhodné se zamyslet nad tím, zda celá konstrukce není zbytečná a problém nelze vyřešit použitím vhodných logických operátorů.
>
> Příkladem je výraz `if` *podmínka* `then True else False`, který se dá zjednodušit na výraz *podmínka*.

Př. 1.2.13 
Zjistěte (bez použití interpretu), na co se vyhodnotí následující výraz. Poté všech sedm funkcí přepište do prefixového tvaru a pomocí interpretu ověřte, že se hodnota výrazu nezměnila.

```
5 + 7 * 5 `mod` 3 `div` 2 == 3 * 2 - 1
```

Př. 1.2.14 

Do následujícího výrazu doplňte implicitní závorky a pak převeďte všechny operátory v něm do prefixového tvaru.

```
2 + 2 * 3 == 2 * 4 && 8 `div` 2 * 2 == 2 || 0 > 7
```

1.3 Definice podle vzoru

Př. 1.3.1 
Definujte s využitím vzorů funkci `isWeekendDay :: String -> Bool`, která rozhodne, jestli je daný řetězec název víkendového dne. Použijte definici podle vzoru.

```
isWeekendDay "Saturday" ⇝* True
isWeekendDay "Monday" ⇝* False
isWeekendDay "apple" ⇝* False
```

Př. 1.3.2 
Definujte funkci `isSmallVowel :: Char -> Bool`, která rozhodne, jestli je dané písmeno malou samohláskou anglické abecedy. Znakové literály se v Haskellu píší do apostrofů, například literál znaku *a* se v Haskellu zapíše jako `'a'`.

> **Jindřiška varuje:** V Haskellu, na rozdíl od například Pythonu (a mnoha dalších skriptovacích jazyků), je rozdíl mezi věcmi uzavřenými mezi apostrofy `'...'` a uvozovkami `"..."`. Věc uzavřená mezi apostrofy je vždy jeden znak (typu `Char`), zatímco mezi uvozovkami se jedná o řetězec (typ `String`; řetězec se skládá ze znaků).

Př. 1.3.3 Definujte funkci `logicalAnd :: Bool -> Bool -> Bool`, která se chová stejně jako funkce logické konjunkce tak, abyste v definici

a) využili podmíněný výraz,
b) nepoužili podmíněný výraz.

Nesmíte využít žádné logické funkce definované v Haskellu.

Př. 1.3.4 Naprogramujte funkci `parallelToAxis`, která o úsečce zadané souřadnicemi bodů v rovině rozhodne, jestli je rovnoběžná s jednou ze souřadnicových os roviny.

Poznámka: Vyhněte se funkcím `fst` a `snd` a použijte výhradně definici podle vzoru.

```
parallelToAxis :: (Integer, Integer) -> (Integer, Integer) -> Bool
parallelToAxis (0, 0) (1, 1) ⇝* False
parallelToAxis (1, 1) (1, 4) ⇝* True
parallelToAxis (16, 42) (12, 19) ⇝* False
parallelToAxis (4, 2) (9, 2) ⇝* True
```

Př. 1.3.5 Vzpomeňte si na příklad 1.2.11 a zkuste jej napsat pomocí vzorů.

1.4 Základní typy

> **Pan Fešák doporučuje:** Příkaz GHCi `:t` je dobrý pro kontrolu, ale je vždy lepší být si schopen určit typy sám. A stejně tak je dobré u funkcí typy vždy psát. Tím si procvičujete přemýšlení v typech a budete moci plně využít toho, že vám je Haskell umí kontrolovat, a může tak poznat rozdíl mezi vaší představou o tom, co má funkce dělat (reflektovanou v typu, který jste napsali), a co skutečně dělá (což je reflektované v typu, který si GHC odvodí).

Př. 1.4.1 Určete typy následujících výrazů a najděte další výrazy stejného typu. Své řešení si ověřte s pomocí interpretu.

a) `'a'`
b) `"Don't Panic."`
c) `not`
d) `(&&)`
e) `True`

Př. 1.4.2 Nalezněte příklady hodnot následujících typů:

a) `Bool`
b) `Integer`
c) `Double`
d) `False`
e) `(Int, Integer)`
f) `(Integer, Double, Bool)`
g) `()`
h) `((), (), ())`

Př. 1.4.3 Určete typy následujících výrazů, zkontrolujte si řešení pomocí interpretu.

a) `True`
b) `"True"`
c) `not True`
d) `True || False`
e) `True && ""`
f) `f 1`, kde funkce `f` je definovaná jako

```
f :: Integer -> Integer
f x = x * x + 2
```

g) `f 3.14`, kde `f` je definovaná stejně jako v části f)
h) `g 3 8`, kde `g` je definovaná jako

```
g :: Int -> Int -> Int
g x y = x * y - 6
```

Př. 1.4.4 Určete typ následujících funkcí, které jsou definovány předpisem:

a)
```
implication a b = not a || b
```

b)
```
foo _   "42" = True
foo 'a' _    = True
foo _   _    = False
```

c)
```
ft True x = False
ft x    y = y
```

*
**

Na konci cvičení byste měli zvládnout:

- pracovat s webovou dokumentací;
- ovládat základní příkazy interpretu GHCi a prakticky je využívat;
- otypovat jednoduché výrazy a funkce;
- vytvářet jednoduché funkce pracující s čísly a pravdivostními hodnotami;
- vytvářet funkce definované podle vzoru;
- prakticky umět využít podmíněný výraz `if` a lokální definice pomocí `let ... in ...` nebo `where`.

Cvičení 2: Rekurze, seznamy, anonymní funkce

Před druhým cvičením je zapotřebí znát:

- zápis funkce definované pomocí vzorů;
- zápis seznamů pomocí výčtu prvků, tj. například `[1, 2, 10]`;
- základní funkce pro práci se seznamy, tj. například

```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
```

- základní vzory pro seznamy, tj. například `[]`, `(x : xs)`, `[x]`, `[x, y]`;
- lokální definice pomocných funkcí pomocí klauzule `where`;
- chování funkcí

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

- zápis anonymních funkcí pomocí λ-abstrakce.

2.1 Rekurze

Př. 2.1.1 »=

Bez použití knihovních funkcí `mod`, `even` a `odd` naprogramujte rekurzivně funkci `isEven`, která pro dané nezáporné celé číslo rozhodne, jestli je sudé. Například:

```
isEven 0 ⇝* True
isEven 3 ⇝* False
isEven 8 ⇝* True
```

Př. 2.1.2 »=

Bez použití knihovní funkce `mod` naprogramujte rekurzivně funkci `mod3`, která pro dané nezáporné celé číslo vypočítá jeho zbytek po dělení 3. Například:

```
mod3 0 ⇝* 0
mod3 5 ⇝* 2
mod3 7 ⇝* 1
mod3 9 ⇝* 0
```

Př. 2.1.3 »=

Bez použití knihovní funkce `div` naprogramujte rekurzivně funkci `div3`, která dané nezáporné celé číslo celočíselně vydělí třemi. Například:

```
div3 0 ⇝* 0
div3 5 ⇝* 1
div3 7 ⇝* 2
div3 9 ⇝* 3
```

Př. 2.1.4 Definujte rekurzivní funkci pro výpočet faktoriálu.

Př. 2.1.5  Definujte funkci `power` takovou, že `power z n` se pro nezáporné číslo `n` vyhodnotí na `z` na `n`-tou. Například:

```
power 3 1 ⇝* 3
power 3 2 ⇝* 9
power 3 3 ⇝* 27
power 3 0 ⇝* 1
power 0 3 ⇝* 0
```

Př. 2.1.6 >>= Napište funkci `isPower2 :: Integer -> Bool`, která o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky. Můžete použít funkce `even` a `odd`. Například:

```
isPower2 0 ⇝* False
isPower2 1 ⇝* True
isPower2 2 ⇝* True
isPower2 3 ⇝* False
isPower2 4 ⇝* True
isPower2 6 ⇝* False
isPower2 8 ⇝* True
```

Př. 2.1.7 >>= Definujte funkci `digits`, která po aplikaci na kladné celé číslo vrátí jeho ciferný součet. Můžete použít funkce `div` a `mod`. Například:

```
digits 123 ⇝* 6
digits 103 ⇝* 4
```

Př. 2.1.8 Napište funkci `mygcd`, která po aplikaci na dvě kladná celá čísla vrátí jejich největšího společného dělitele. Pokuste se o co nejefektivnější implementaci.

Př. 2.1.9 Naprogramujte funkci `primeDivisors :: Integer -> Integer`, která rozhodne, součinem kolika prvočísel je zadané kladné číslo. Můžete použít funkce `div` a `mod`. Například:

```
primeDivisors 3  ⇝* 1        primeDivisors 8  ⇝* 3
primeDivisors 4  ⇝* 2        primeDivisors 12 ⇝* 3
primeDivisors 5  ⇝* 1        primeDivisors 15 ⇝* 2
primeDivisors 6  ⇝* 2        primeDivisors 1  ⇝* 0
```

Př. 2.1.10 Definujte funkce `plus` a `times`, které budou ekvivalentní operátorům `(+)` a `(*)` na přirozených číslech. Je zakázáno v implementaci používat vestavěné funkce `(+)` a `(*)`. Můžete však používat libovolné jiné funkce, doporučujeme podívat se zejména na funkce `pred` a `succ` (jejich typ je ve skutečnosti o něco obecnější, ale můžete uvažovat, že to je `Integer -> Integer`).

Bonus: implementujte funkce `plus'` a `times'`, které budou fungovat na všech celých číslech.

Př. 2.1.11  Co počítá následující funkce? Jak se chová na argumentech, kterými jsou nezáporná čísla? Jak se chová na záporných argumentech?

```
fun 0 = 0
fun n = fun (n - 1) + 2 * n - 1
```

2.2 Seznamy

Př. 2.2.1 Určete typy seznamů:

a) `["a", "b", "c"]`
b) `['a', 'b', 'c']`
c) `"abc"`
d) `[(True, ()), (False, ())]`
e) `[(++) "abc" "def", "X" ++ "Y" ++ "Z"]`
f) `[(&&), (||)]`
g) `[]`
h) `[[]]`
i) `[[], [True]]`

Př. 2.2.2 Napište funkci, která na začátek zadaného seznamu čísel vloží číslo `42`. Jaký má vaše funkce typ?

Př. 2.2.3 Bez použití jakýchkoli knihovních funkcí definujte funkci `isEmpty` typu `[a] -> Bool`, která pro prázdný vstupní seznam vrátí `True` a pro neprázdný vrátí `False`.

Př. 2.2.4 Bez použití funkcí `head` a `tail` napište funkci `myHead` typu `[a] -> a`, která vrátí první prvek zadaného neprázdného seznamu, a funkci `myTail` typu `[a] -> [a]`, která pro zadaný neprázdný seznam vrátí tentýž seznam bez prvního prvku.

Př. 2.2.5 Definujte funkci `second :: [a] -> a`, která vrátí druhý prvek zadaného seznamu. Můžete předpokládat, že vstupní seznam obsahuje alespoň dvě čísla.

Př. 2.2.6 Pro následující vzory a seznamy určete, které vzory mohou reprezentovat které seznamy. Stanovte, jak se navážou proměnné ze vzoru.

- Vzory: `[]`, `x`, `[x]`, `[x, y]`, `(x : s)`, `(x : y : s)`, `[x : s]`, `((x : y) : s)`
- Seznamy: `[1]`, `[1, 2]`, `[1, 2, 3]`, `[[]]`, `[[1]]`, `[[1], [2, 3]]`

Př. 2.2.7 Bez použití knihovní funkce `last` definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu.

Př. 2.2.8 Bez použití funkce `init` definujte funkci `stripLast` typu `[a] -> [a]`, která pro neprázdný seznam vrátí tentýž seznam bez posledního prvku.

Př. 2.2.9 Bez použití knihovní funkce `length` definujte funkci `len :: [a] -> Integer`, která spočítá délku zadaného seznamu.

Př. 2.2.10 Napište funkci `containsNumber :: Integer -> [Integer] -> Bool`, která vrací `True`, pokud je první argument obsažen v seznamu zadaném druhým argumentem, jinak vrací `False`. Například:

```
containsNumber 42 [1, 2, 3] ⇝* False
containsNumber 2 [1, 2, 3] ⇝* True
containsNumber 2 [] ⇝* False
```

Př. 2.2.11 Napište funkci `containsNNumbers` typu `Integer -> Integer -> [Integer] -> Bool` takovou, že `containsNNumbers n x xs` bude `True` právě tehdy, když seznam `xs` obsahuje alespoň `n` výskytů čísla `x`. Například:

```
containsNNumbers 2 42 [1, 2, 42] ⇝* False
containsNNumbers 2 42 [1, 42, 2, 42] ⇝* True
containsNNumbers 3 42 [1, 42, 2, 42] ⇝* False
containsNNumbers 0 42 [1, 2] ⇝* True
```

Př. 2.2.12 Mějme neprázdný seznam typu `[(String, Integer)]`, který reprezentuje seznam jmen studentů s jejich počty bodů z předmětu IB015. Naprogramujte

a) funkci `getPoints :: String -> [(String, Integer)] -> Integer`, která vrátí počet bodů studenta se jménem zadaným v prvním argumentu (nebo 0, pokud takový student v seznamu není),

b) funkci `getBest :: [(String, Integer)] -> String`, která vrátí jméno studenta s nejvíce body.

Například tedy:

```
getPoints "Stan" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* 20
getPoints "Tomas" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* 0
getBest [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* "Eric"
```

Př. 2.2.13 Napište funkci `nth :: Integer -> [a] -> a`, která ze zadaného seznamu vrátí prvek na pozici určené prvním argumentem funkce. Můžete předpokládat, že vstupní seznam má dostatečnou délku. Například:

```
nth 0 [4, 3, 5] ⇝* 4
nth 1 [4, 3, 5] ⇝* 3
nth 2 [4, 3, 5] ⇝* 5
nth 2 "Get Schwifty" ⇝* 't'
```

Př. 2.2.14 Napište funkci `append :: [a] -> [a] -> [a]`, jejíž výsledek pro dva seznamy bude seznam, který vznikne zřetězením těchto dvou seznamů. Nepoužívejte knihovní operátor `(++)`. Například:

```
append [1, 2] [3, 4, 8] ⇝* [1, 2, 3, 4, 8]
append [] [3, 4] ⇝* [3, 4]
append [3, 4] [] ⇝* [3, 4]
append "Legen" "dary" ⇝* "Legendary"
```

Př. 2.2.15 Napište funkci `pairs :: [a] -> [(a, a)]` , která bere prvky ze vstupního seznamu po dvou prvcích a vytváří seznam dvojic těchto prvků. Pokud má seznam lichý počet prvků, poslední prvek se zahodí. Například:

```
pairs [4, 8, 15, 16, 23] ⇝* [(4, 8), (15, 16)]
pairs [4, 8, 15, 16, 23, 42] ⇝* [(4, 8), (15, 16), (23, 42)]
pairs "Humphrey" ⇝* [('H', 'u'), ('m', 'p'), ('h', 'r'), ('e', 'y')]
```

Př. 2.2.16 Napište následující funkce pracující se seznamy čísel pomocí rekurze a vzorů:

a) `listSum :: [Integer] -> Integer`, která dostane seznam čísel a vrátí součet všech jeho prvků.

b) `oddLength :: [Integer] -> Bool`, která vrátí `True`, pokud je seznam liché délky, jinak `False` (bez použití funkce `length`).

c) `add1 :: [Integer] -> [Integer]`, která každé číslo ve vstupním seznamu zvýší o 1,

d) `multiplyN :: Integer -> [Integer] -> [Integer]`, která každé číslo ve vstupním seznamu vynásobí prvním argumentem funkce,

e) `deleteEven :: [Integer] -> [Integer]`, která ze seznamu čísel odstraní všechna sudá čísla,
f) `deleteElem :: Integer -> [Integer] -> [Integer]`, která ze seznamu čísel odstraní všechny výskyty čísla zadaného prvním argumentem,
g) `largestNumber :: [Integer] -> Integer`, vrátí největší číslo ze zadaného neprázdného seznamu čísel,
h) `listsEqual :: [Integer] -> [Integer] -> Bool`, která dostane na vstup dva seznamy a vrátí `True` právě tehdy, když se rovnají (bez použití funkce `(==)` na seznamy),
i) `multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2 (lichá čísla vynechá),
j) `sqroots :: [Double] -> [Double]`, která ze zadaného seznamu vybere kladná čísla a ta odmocní (může se vám hodit funkce `sqrt`).

Př. 2.2.17 Napište funkci `everyNth :: Integer -> [a] -> [a]` takovou, že seznam `everyNth n xs` bude obsahovat každý n-tý prvek ze seznamu `xs`. Například:


```
everyNth 2 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 1, 2, 7]
everyNth 3 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 3, 7]
everyNth 4 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 2]
everyNth 1 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 8, 1, 3, 2, 5, 7]
everyNth 2 "BoJack Horseman" ⇝* "BJc osmn"
```

Př. 2.2.18 Napište funkci `brackets :: String -> Bool`, která dostane řetězec složený ze znaků `'('` a `')'` a rozhodne, jestli se jedná o korektní uzávorkování. Například:


```
brackets "(()())" ⇝* True
brackets "(()()" ⇝* False
brackets "())()" ⇝* False
brackets "())(" ⇝* False
brackets "" ⇝* True
```

Př. 2.2.19 Zadefinujte funkci `palindrome`, která na vstupu dostane řetězec a rozhodne o něm, jestli je palindrom. Napište druhou funkci `palindromize`, která ze zadaného řetězce udělá palindrom tak, že na jeho konec doplní co nejméně znaků. Tedy například:


```
palindrome "brienne" ⇝* False
palindromize "brienne" ⇝* "brienneirb"
```

Př. 2.2.20 Napište funkci `getMiddle :: [a] -> a`, která pro zadaný neprázdný seznam vrátí jeho prostřední prvek bez zjišťování jeho délky. Pokud má seznam sudý počet prvků, vraťte levý z prostředních dvou. Například:

```
getMiddle [1] ⇝* 1
getMiddle [2, 1] ⇝* 2
getMiddle [2, 1, 5] ⇝* 1
getMiddle [2, 1, 5, 6] ⇝* 1
getMiddle [2, 1, 5, 6, 3] ⇝* 5
getMiddle "Don't blink!" ⇝* ' '
```

Nápověda: Pokud zajíc běží dvakrát rychleji než želva, pak v okamžiku, kdy zajíc vyhrál závod, je želva v polovině trati.

2.3 Užitečné seznamové funkce

> **Pan Fešák doporučuje:** Pokud si nejste chováním některé funkce jistí, můžete ji najít v **dokumentaci**. Teď se například může hodit najít si seznamové funkce jako `map` a `filter`.

Př. 2.3.1 Slovně vysvětlete, co dělají knihovní funkce `map` a `filter`. Pro jaký účel byste použili kterou z nich? Jaký je rozdíl mezi výsledkem vyhodnocení `map even [1, 3, 2, 5, 4, 8, 11]` a `filter even [1, 3, 2, 5, 4, 8, 11]`?

Př. 2.3.2 Mějme seznam typu `[(String, Integer)]`, který obsahuje jména studentů a jejich počty bodů z předmětu IB015. Pomocí vhodných seznamových funkcí naprogramujte

a) funkci `getNames :: [(String, Integer)] -> [String]`, která vrátí seznam jmen studentů,

b) funkci `successfulRecords :: [(String, Integer)] -> [(String, Integer)]`, která ze zadaného seznamu vybere záznamy těch studentů, kteří mají alespoň 50 bodů,

c) funkci `successfulNames :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere jména studentů, kteří mají alespoň 50 bodů,

d) funkci `successfulStrings :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere studenty, kteří mají alespoň 50 bodů, a vrátí seznam řetězců ve tvaru `"jmeno: xxx b"` (*nápověda*: pro převod čísla na řetězec můžete použít funkci `show`).

Tedy například pro databázi

```
st :: [(String, Integer)]
st = [("Finn", 35), ("Jake", 53), ("Bubblegum",  98),
      ("Ice King", 20), ("BMO", 99)]
```

budou požadované funkce vracet následující hodnoty:

```
getNames st ⇝* ["Finn", "Jake", "Bubblegum", "Ice King", "BMO"]
successfulRecords st ⇝* [("Jake", 53), ("Bubblegum",  98), ("BMO", 99)]
successfulNames st ⇝* ["Jake", "Bubblegum", "BMO"]
successfulStrings st ⇝* ["Jake: 53 b", "Bubblegum: 98 b", "BMO: 99 b"]
```

Př. 2.3.3 Které z funkcí z příkladu 2.2.16 lze elegantně naprogramovat pomocí funkce `map`? Které lze elegantně naprogramovat pomocí funkce `filter`? Všechny tyto funkce pomocí `map` a `filter` naprogramujte.

Př. 2.3.4 S využitím funkce `map` a knihovní funkce `toUpper :: Char -> Char` z modulu `Data.Char` (tj. je třeba použít `import Data.Char`, na začátku souboru, nebo `:m +Data.Char` v interpretu) definujte novou funkci `toUpperStr`, která převádí řetězec písmen na řetězec velkých písmen. Například:

```
toUpperStr "i am the one who knocks!" ⇝* "I AM THE ONE WHO KNOCKS!"
```

Př. 2.3.5 Napište funkci `vowels`, která dostane seznam řetězců a vrátí seznam řetězců takových, že v každém řetězci ponechá jenom samohlásky (ale zachová jejich pořadí). Například:

```
vowels ["Michael", "Dwight", "Jim", "Pam"] ⇝* ["iae", "i", "i", "a"]
vowels ["MICHAEL", "DWIGHT", "JIM", "PAM"] ⇝* ["IAE", "I", "I", "A"]
```

Př. 2.3.6 Slovně vysvětlete, co dělají funkce `zip` a `zipWith`. Pro jaký účel byste použili kterou z nich? Pomocí interpretu zjistěte, jak se tyto funkce chovají, pokud mají vstupní seznamy různou délku.

Př. 2.3.7 Mějme výsledky běžeckého závodu reprezentované pomocí seznamu typu `[String]`, který obsahuje jména běžců seřazených od nejlepšího po nejhoršího, a seznam peněžních výher typu `[Integer]`. Naprogramujte

a) funkci `assignPrizes` typu `[String] -> [Integer] -> [(String, Integer)]`, která každému běžci, který něco vyhrál, přiřadí jeho výhru, a
b) funkci `prizeTexts` typu `[String] -> [Integer] -> [String]`, která vrátí seznam řetězců ve tvaru `"jmeno: xxx Kc"` pro každého běžce, který něco vyhrál.

Například:

```
assignPrizes ["Mike", "Dustin", "Lucas", "Will"] [100, 50] ⇝*
    [("Mike", 100), ("Dustin", 50)]
prizeTexts ["Mike", "Dustin", "Lucas", "Will"] [100, 50] ⇝*
    ["Mike: 100 Kc", "Dustin: 50 Kc"]
```

Př. 2.3.8 Pomocí funkce `zip` napište funkci `neighbors :: [a] -> [(a, a)]`, která pro zadaný seznam vrátí seznam dvojic sousedních prvků. Například:

```
neighbors [3, 8, 2, 5] ⇝* [(3, 8), (8, 2), (2, 5)]
neighbors [3, 8] ⇝* [(3, 8)]
neighbors [3] ⇝* []
neighbors "Kree!" ⇝* [('K', 'r'), ('r', 'e'), ('e', 'e'), ('e', '!')]
```

Př. 2.3.9 Napište funkci, která zjistí, jestli jsou v seznamu čísel některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

Př. 2.3.10 Implementujte funkce `myMap`, `myFilter` a `myZipWith`, které se budou chovat jako knihovní funkce `map`, `filter` a `zipWith`.

Př. 2.3.11 Uvažte funkci `anyEven :: [Integer] -> Bool`, která rozhodne, jestli je v seznamu čísel nějaké kladné číslo, a funkci `allEven :: [Integer] -> Bool`, která rozhodne, jestli jsou všechna čísla v seznamu kladná. Najděte ve standardní knihovně funkci nebo funkce, pomocí kterých lze funkce `anyEven` a `allEven` implementovat jednoduše bez explicitního použití rekurze.

Př. 2.3.12 Zjistěte, co dělají funkce `takeWhile` a `dropWhile`.

2.4 λ-abstrakce

Př. 2.4.1 Slovně popište, co dělají následující funkce:

a) `\x -> 4 * x + 2`
b) `\x y -> x + 2 * y`
c) `\(x, y) -> x + y`
d) `\x y -> x`
e) `\(x, y) -> x`

Př. 2.4.2 Implementujte funkce z příkladu 2.3.2 tak, že místo vlastních pomocných funkcí použijete λ-abstrakci.

Př. 2.4.3 Implementujte všechny funkce ze sekce 2.3 tak, že místo vlastních pomocných funkcí použijete λ-abstrakci.

Př. 2.4.4 
Pomocí rekurze a funkce `filter` napište funkci `quickSort :: [Integer] -> [Integer]`, která seřadí vstupní seznam vzestupně pomocí algoritmu *quick sort*. Například:

```
quickSort [5, 3, 8, 12, 1] ⇝* [1, 3, 5, 8, 12]
quickSort [5, 4, 3, 2] ⇝* [2, 3, 4, 5]
quickSort [2, 2, 2] ⇝* [2, 2, 2]
quickSort [2] ⇝* [2]
quickSort [] ⇝* []
```

Pokud algoritmus quick sort neznáte, zkuste ho nastudovat například na Wikipedii.

Na konci druhého cvičení byste měli umět:

- definovat vlastní rekurzivní funkce pracující s celými čísly;
- pracovat se seznamy a definovat na nich funkce pomocí vzorů;
- definovat rekurzivní funkce na seznamech;
- poznat, kdy je na práci se seznamy vhodné použít knihovní funkce `map`, `filter`, `zip` a `zipWith`, a umět tyto funkce použít;
- poznat, kdy je vhodné použít λ-abstrakci, a umět pomocí λ-abstrakce definovat jednoduché funkce.

Cvičení 3: Manipulace s funkcemi, typy

Před třetím 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 funkcí;
- co je částečná aplikace;
- základní funkce pro manipulace s čísly a seznamy.

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.

3.1 Typy a typové třídy

Př. 3.1.1 Určete typy výrazů:


a) `not True`
b) `(&&)`
c) `[]`
d) `"Don't Panic!"`
e) `[(True, ""), (False, [])]`
f) `[True, []]`

Pan Fešák doporučuje: Informace o typových třídách a v nich definovaných funkcích můžeme zjistit pomocí příkazu `:i` v GHCi. Např.

```
> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
        -- Defined in ‘GHC.Real’
instance Fractional Float -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
```

To nám říká, že typová třída `Fractional` obsahuje funkce `(/)`, `recip` a `fromRational`, že do ní patří typy `Float` a `Double` a že je-li něco ve třídě `Fractional`, pak je to nutně také ve třídě `Num`. Krom toho nám tento výpis také

dává informace o tom, které funkce bychom minimálně museli implementovat, pokud bychom chtěli do této třídy nějaký nový typ přidat.

Chceme-li zjistit, zda jde nějaký kontext, například `(Num a, Floating a)` zjednodušit, potřebujeme zjistit, zda některá z těchto typových tříd vyžaduje některou další. V tomto případě bude potřeba navíc jít přes mezikrok:

```
class Fractional a => Floating a where {- ... -}
class Num a => Fractional a where {- ... -}
```

Tedy pokud je nějaký typ ve `Floating`, musí být nutně ve `Fractional` a tedy i v `Num`. Kontext tedy zjednodušíme na `Floating a`.

Př. 3.1.2 Určete typy výrazů. Snažte se neuvádět typové třídy, které jsou již implikovány jinými typovými třídami. Můžete použít interpret k zjištění typů knihovních funkcí a závislostí mezi typovými třídami.

a) `[1, 2, 3.14]`
b) `head [2 ^ 2]`
c) `[1, 2, True]`
d) `filter (\x -> x > 5) [1, 2, 4, 8]`
e) `\x -> show (x ^ x)`
f) `\x -> read x + 2`
g) `\x -> (fromIntegral x :: Double)`

Př. 3.1.3 Určete typy následujících funkcí:

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

Pan Ešák připomíná: V případě, že chci otypovat funkci definovanou víceřádkovou definicí, otypuji každý řádek zvlášť a pak unifikuji typy argumentů na odpovídajících pozicích a typy návratových hodnot.

Př. 3.1.4 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. 3.1.5 Určete typy následujících funkcí a popište slovně, co funkce dělají.

a)
```
cm _ []       = []
cm f (x : xs) = f x ++ cm f xs
```
b)
```
mm [] = error "empty list"
mm (x0 : xs0) = mm' x0 x0 xs0
    where
        mm' a b []       = (a, b)
        mm' a b (x : xs) = mm' (min a x) (max b x) xs
```

Př. 3.1.6 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`

3.2 Částečná aplikace a operátorové sekce

Př. 3.2.1 Vysvětlete, co dělají následující funkce, a najděte argumenty, na něž je lze aplikovat. Následně si chování ověřte v GHCi.

a) `(+) 2`
b) `(* 2)`
c) `(- 2)`
d) `(2 -)`
e) `(/ 2)`

Př. 3.2.2 Přepište v následujících definicích seznamových funkcí lambda funkce na částečnou aplikaci tak, aby funkčnost zůstala stejná.

a)
```
import Data.Char
upper :: String -> String
upper xs = map (\x -> toUpper x) xs
```

b)
```
embrace :: [String] -> [String]
embrace xs = map (\x -> '[' : x) (map (\x -> x ++ "]") xs)
```

c)
```
sql :: (Ord a, Num a) => [a] -> a -> [a]
sql xs lt = map (\x -> x ^ 2) (filter (\x -> x < lt) xs)
```

Př. 3.2.3 Které z následujících výrazů jsou ekvivalentní?

a) `f 1 g 2` $\stackrel{?}{\equiv}$ `f 1 (g 2)`
b) `(f 1 g) 2` $\stackrel{?}{\equiv}$ `(f 1) g 2`
c) `(* 2) 3` $\stackrel{?}{\equiv}$ `2 * 3`
d) `(*) 2 3` $\stackrel{?}{\equiv}$ `(* 3) 2`

3.3 Skládání funkcí, η-redukce, odstraňování argumentů

> **Pan Fešák vysvětluje:** Řecké písmeno η je éta (a jeho ocásek se píše pod linku, stejně jako například u našeho j). Pojem η-redukce pochází z Lambda kalkulu a představuje odstraňování formálních argumentů. Někdy se můžete setkat také s pojmem η-konverze, který představuje jak odebírání argumentů, tak i jejich přidávání (tam, kde to typ dovoluje). Písmeno η bylo vybráno kvůli souvislosti s *extensionalitou*, tedy s tvrzením $f = g \iff \forall x.(f(x) = g(x))$.

> **Pan Fešák vysvětluje:** Odstraněním argumentů z funkce ji převedeme na *pointfree* tvar, naopak pokud funkce má v definici všechny argumenty, o nichž hovoří její typ, je v *pointwise* tvaru. Onen *point* v těchto názvech představuje argument funkce (bod), nikoli tečku (skládání funkcí), více naleznete na Haskell Wiki. Mezi těmito tvary lze vždy převádět, někdy to však není vhodné či snadné, jednak kvůli čitelnosti, jednak je obtížné odstranit argument, který je v definici funkce použit vícekrát.

> **Jindřiška varuje:** Nic se nesmí přehánět, funkce jako `(.(,)) . (.) . (,)` nebo `(.) . (.)` nejsou ani hezké ani čitelné.

Př. 3.3.1 Otypujte následující výrazy:

a) `map even`
b) `map head . snd`
c) `filter ((4 >) . maximum)`
d) `const const`

Př. 3.3.2 Přepište v následujících definicích seznamových funkcí lambda funkce pomocí skládání funkcí, částečné aplikace nebo operátorové sekce tak, aby funkčnost zůstala stejná. Odstraňte také formální argumenty funkcí, pokud to je smysluplné.

a)
```
failing :: [(Int, Char)] -> [Int]
failing sts = map fst (filter (\t -> snd t == 'F') sts)
```

b)
```
embraceWith :: Char -> Char -> [String] -> [String]
embraceWith l r xs = map (\x -> l : x ++ [r]) xs
```

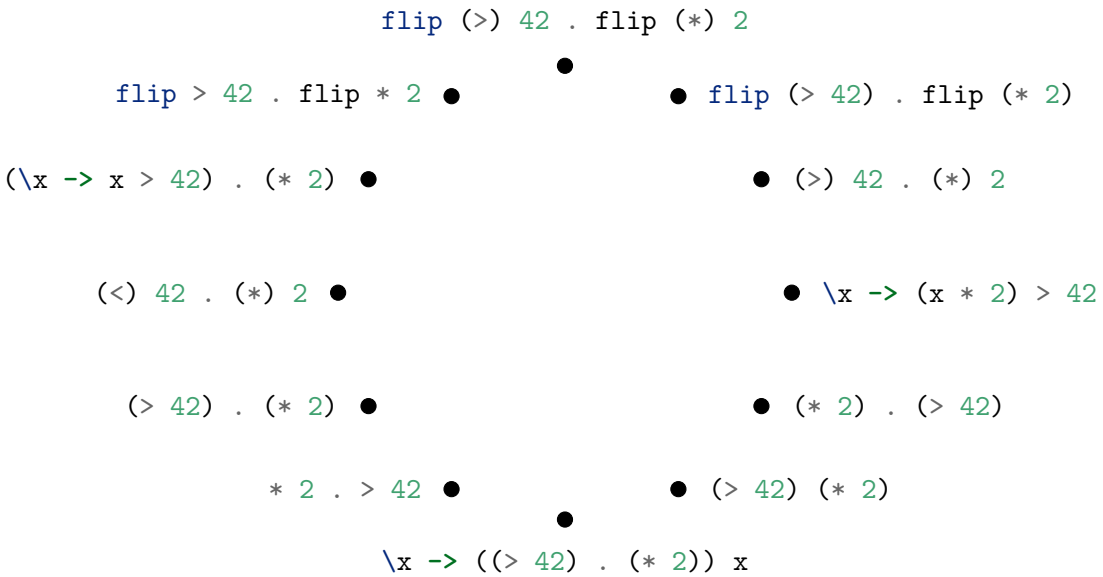
(`l` a `r` neodstraňujte)

c)
```
divisibleBy7 :: [Integer] -> [Integer]
divisibleBy7 xs = filter (\x -> x `mod` 7 == 0) xs
```

d)
```
import Data.Char
letterCaesar :: String -> String
letterCaesar xs = map (\x -> chr (3 + ord x)) (filter isLetter xs)
```

e)
```
zp :: (Integral a, Num b) => [a] -> [b] -> [b]
zp xs ys = zipWith (\x y -> y ^ x) xs ys
```

Př. 3.3.3 Mezi následujícími výrazy najděte všechny korektní a mezi nimi rozhodněte, které jsou vzájemně ekvivalentní (vzhledem k chování na libovolných vstupech povolených typem výrazu). Zdůvodněte neekvivalenci.



Př. 3.3.4 Uvažme funkci `negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních predikátů (funkcí typu `a -> Bool`). Tj. funkce `negp pred` vrátí opačnou logickou hodnotu, než by vrátil predikát `pred` na zadané hodnotě. Tedy například `negp even` by mělo být ekvivalentní s `odd`.

a) Definujte funkci `negp` (můžete využít třeba funkci `not`).
b) Definujte funkci `negp` jako unární funkci (s použitím pouze jednoho formálního parametru).
c) Definujte funkci `negp` bez použití formálních parametrů.

Př. 3.3.5 Pokud to je možné, přepište lambda funkce v následujících definicích pomocí skládání funkcí, částečné aplikace nebo operátorové sekce tak, aby funkčnost zůstala stejná. Odstraňte také formální argumenty funkcí.

```
import Data.Char

-- | Convert lowercase letters to numbers 0..25
l2c :: Char -> Int
l2c c = ord c - ord 'a'

-- | Convert 0..25 character codes to uppercase letters
c2l :: Int -> Char
c2l c = chr (c + ord 'A')

-- | Keep only lowercase English letters
lowAlphaOnly :: String -> String
lowAlphaOnly xs = filter (\x -> isLower x && isAscii x) xs

-- | Encrypt messages using Vigenere (one-time-pad) cipher
letterVigenere :: String -> String -> String
letterVigenere xs ks = zipWith
    (\x y -> c2l ((l2c x + l2c y) `mod` 26))
    (lowAlphaOnly xs)
    (lowAlphaOnly ks)
```

Nápověda: formální argument nelze odstranit (s tím, co jsme se učili), pokud je v definici použit vícekrát.

K odstranění formálního argumentu v použitého vícekrát v těle funkce lze použít funkci `(<*>)`. Její typ je sice velice obecný (a komplikovaný), ale pro tyto účely ji můžeme otypovat jako `(<*>) :: (a -> b -> c) -> (a -> b) -> a -> c`. Rovněž ji pro tyto účely můžeme nahradit následující funkcí:

```
dist :: (a -> b -> c) -> (a -> b) -> a -> c
dist f g x = f x (g x)
```

Př. 3.3.6 Převeďte následující funkce do pointfree tvaru, neboli odstraňte formální argumenty lambda abstrakcí:

a) `\x -> (f . g) x`
b) `\x -> f . g x`
c) `\x -> f x . g`

Př. 3.3.7 Převeďte následující výrazy do pointwise tvaru, neboli přidejte všechny argumenty, které plynou z typu výrazu:

a) `(^ 2) . mod 4 . (+ 1)`

b) `(+) . sum . take 10`
c) `map f . flip zip [1, 2, 3]` (funkce `f` je definována externě)
d) `(.)`

Př. 3.3.8 Určete typ následujících funkcí. Přepište tyto definice funkcí tak, abyste v jejich definici nepoužili λ-abstrakci a formální parametry (tj. chce se pointfree definice).

> **Pan Fešák vysvětluje:** Pokud potřebuji odstranit formální parametr, jenž se nevyskytuje v těle výrazu, pomůžu si funkcí `const`: pro libovolný výraz `w`, který nepřidává žádná nová typová omezení, je výraz `v` ekvivalentní s výrazem `const v w`. Tento výraz už obsahuje v těle navíc parametr `w`, který se dá použít pro η-redukci.

a) `f x y = y`
b) `f x y = 3 + x`

Př. 3.3.9 Převeďte následující funkce do pointfree tvaru:

a) `\_ -> x`
b) `\x -> f x 1`
c) `\x -> f 1 x True`
d) `const x`
e) `\x -> 0`
f) `\x -> if x == 1 then 2 else 0`
g) `\f -> flip f x`

Př. 3.3.10 Převeďte všechny níže uvedené funkce do pointfree tvaru. Při převodu třetí si pomozte převodem druhé.

a) `f1 x y z = x`
b) `f2 x y z = y`
c) `f3 x y z = z`

Př. 3.3.11 Zapište v pointfree tvaru funkci `g x = f x c1 c2 c3 ... cn` (`f` je nějaká pevně daná funkce a `c1`, `c2`, ..., `cn` jsou konstanty).

*
**

Na konci cvičení byste měli zvládnout:

- otypovat výrazy a funkce, a to včetně polymorfních funkcí využívajících typové třídy a včetně funkcí definovaných podle vzoru;
- Na intuitivní úrovni znát význam typových tříd `Num`, `Integral`, `Fractional`, `Show`, `Read`;
- používat operátorové sekce a částečnou aplikaci;
- skládat funkce a pochopit kód obsahující skládání funkcí;
- odstranit formální parametry funkce (pokud je každý použit nejvýše jednou);
- z typu poznat, kolik argumentů funkce má, a umět je přidat do její definice a odstranit při tom skládání funkcí a nyní již zbytečné částečné aplikace.

Cvičení 4: Vlastní a rekurzivní datové typy, `Maybe`

Před čtvrtým cvičením je zapotřebí:

- znát koncept datových typů:
  - *hodnotový* a *typový* konstruktor;
  - klíčové slovo `data`;
  - definice funkcí pomocí vzorů pro vlastní datové typy;
- umět pro vlastní datový typ podmínky k implementaci jednoduché instance typových tříd;
- znát datový typ `Maybe`;
- mít základní znalosti o *stromech* – pojmy **kořen**, **cesta**, **hloubka vrcholu**.

4.1 Vlastní datové typy

Př. 4.1.1 Mějme následující definici:

```
data Object = Cube Float Float Float   -- a, b, c
            | Cylinder Float Float     -- r, v
```

a) Uveďte hodnoty, které mají typ `Object`?
b) Kolik je v definici použito hodnotových konstruktorů a které to jsou?
c) Kolik je v definici použito typových konstruktorů a které to jsou?
d) Definujte funkce `volume` a `surface`, které pro hodnoty uvedeného typu počítají objem, respektive povrch.
Příklady vyhodnocení korektně definovaných funkcí jsou:
`volume (Cube 1 2 3)` $\rightsquigarrow^*$ `6`
`surface (Cylinder 1 3)` $\rightsquigarrow^*$ `25.132741228718345`

Př. 4.1.2 Mějme datový typ `Day` představující dny v týdnu definovaný níže. Definujte funkci `weekend :: Day -> Bool`, která o zadaném dni určí, jestli je to víkendový den. Datový typ `Day` je definován takto:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
         deriving (Show, Eq, Ord)
```

Př. 4.1.3 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. 4.1.4 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`.

Př. 4.1.5 Zadefinujme vlastní typ uspořádaných dvojic s názvem `PairT`. Tento typ bude mít pouze jeden binární datový 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. 4.1.6 Vytvořte nový datový typ `Jar` představující sklenici ve spíži. Každá sklenice je v jednom z následujících stavů:

- je prázdná (`EmptyJar`);
- je v ní ovocná marmeláda (`Jam`), pamatujeme si typ ovoce, ze kterého byla vyrobena (`String`);
- jsou v ní okurky (`Cucumbers`), o nich si nemusíme nic pamatovat, stejně se hned snědí;
- je v ní kompot (`Compote`), pamatujeme si rok výroby (`Int`).

Vaší úlohou je pak nadefinovat funkci `stale :: Jar -> Bool`, která určí, jestli je obsah dané sklenice již zkažený. Prázdné sklenice, okurky ani marmelády se nekazí (možná je to tím, že se příliš rychle snědí), kompoty se pokazí za 10 let od zavaření (zadefinujte si celočíselnou konstantu `today`, ve které budete mít aktuální rok).

> **Pan Fešák doporučuje:** Pro úplné pochopení principů vlastních datových typů a rozdílů mezi hodnotovými a typovými konstruktory je doporučené projít a rozumět následujícím příkladům.

Př. 4.1.7 Identifikujte nově vytvořené typové a hodnotové konstruktory a určete jejich aritu.

a) `data X = Value Int`

b)
```
data M = A | B | N M
data N = C | D | M N
```

c) `data Ha = Hah Int Float [Hah]`

d) `data FMN = T (Int, Int) (Int -> Int) [Int]`

e) `type Fat = Float -> Float -> Float`

f) `data E = E (E, E)`

Př. 4.1.8 Které deklarace datových typů jsou správné?

a) `data N x = NVal (x -> x)`

b) `type Makro = a -> a`

c) `data M = N (x, x) | N Bool | O M`

d) `type Fun a = a -> (a, Bool) -> c`

e) `type Fun (a, c) (a, b) = (b, c)`

f) `data F = X Int | Y Float | Z X`

g) `data F = intfun Int`

h) `data F = Makro Int -> Int`

i) `type Val = Int | Bool`
j) `data X = X X X`

Př. 4.1.9 

Uvažte datový typ `type Frac = (Int, Int)`, kde hodnota `(a, b)` představuje zlomek $\frac{a}{b}$ (můžete předpokládat, že $b \neq 0$). Napište funkci nad datovým typem `Frac`, která

a) zjistí, jestli zadané dva zlomky představují stejné racionální číslo;
b) vrátí `True`, jestli zlomek představuje nezáporné číslo;
c) vypočítá součet dvou zlomků;
d) vypočítá rozdíl dvou zlomků;
e) vypočítá součin dvou zlomků;
f) vypočítá podíl dvou zlomků (ověřte, že druhý zlomek je nenulový);
g) převede zlomek do základního tvaru; *Doporučujeme:* zkuste si najít v dokumentaci něco o funkci `gcd`.

Když budete mít všechno implementované, tak upravte funkce tak, aby byl výsledek v základním tvaru.

Př. 4.1.10 

V řetězci kaváren StarBugs prodávají jediný druh šálků kávy. Obyčejní zákazníci platí 13 λ za šálek kávy a každý desátý šálek mají zdarma. Pokud si kávu kupuje zaměstnanec, má navíc 15% slevu ze základní ceny. V případě, že si kávu kupuje student ve zkouškovém období, platí za každý šálek 1 λ. Napište funkci, která spočítá výslednou cenu v závislosti na typu zákazníka. Ale pozor! Slevový systém se často mění, aby zaujal lidi. Proto je potřeba navrhnout funkci dostatečně obecně, aby se nemusela vždy celá přepisovat.

- Napište funkci `commonPricing :: Int -> Float`, která na základě počtu vypitých šálků spočítá cenu pro běžného zákazníka.
- Napište funkci `employeeDiscount :: Float -> Float`, která aplikuje zaměstnaneckou slevu na cenu pro obyčejné zákazníky.
- Napište funkci `studentPricing :: Int -> Float`, která na základě počtu vypitých šálků spočítá cenu pro studenta.
- Definujte datový typ `PricingType`, který bude značit, zdali je nakupující obyčejný zákazník (`Common`), zaměstnanec řetězce (`Employee`) nebo student (`Student`).
- Implementujte funkci

  ```
  computePrice :: PricingType -> Int -> (Int -> Float)
               -> (Float -> Float) -> (Int -> Float) -> Float
  ```

  Ta podle typu zákazníka, počtu šálků a tří funkci `cp`, `ed` a `sp` (`common pricing`, `employee discount` a `student pricing`) spočítá výslednou cenu za nakoupené šálky.

Při řešení se vám může hodit funkce `fromIntegral`. Více se o ní můžete dočíst v dokumentaci.

Příklady vstupů a odpovídajících výsledků:

```
computePrice Common   28 commonPricing employeeDiscount studentPricing
    ⇝* 338
computePrice Employee 28 commonPricing employeeDiscount studentPricing
    ⇝* 287.30002
computePrice Student  28 commonPricing employeeDiscount studentPricing
    ⇝* 28
```

> Následující příklad je rozšířením předchozí úlohy. Doporučujeme vrátit se k němu, pokud máte na konci cvičení čas.

Př. 4.1.11 

Řetězec StarBugs z úlohy 4.1.10 se rozhodl rozšířit svůj systém slev. Ještě neví jak přesně, ale každá cena bude buď závislá na počtu koupených šálků, nebo na běžné ceně pro obyčejné zákazníky za daný počet šálků.

Upravte datový typ `PricingType` tak, aby nabízel možnosti `Common` (běžný zákazník), `Special (Int -> Float)` (speciální druh nacenění závislý na počtu káv) a `Discount (Float -> Float)` (slevový druh nacenění závislý na běžné ceně). Každá instance tohoto typu (krom `Common`) tak v sobě bude nést funkci pro výpočet správné ceny.

Dále je nezbytné změnit i funkci `computePrice`, a to tak, že její typ bude `PricingType -> Int -> (Int -> Float) -> Float`. Akceptuje druh zákazníka, počet šálků a funkci pro výpočet běžné ceny a vrací správnou cenu za příslušný počet šálků.

Jako poslední definujte konstanty `common`, `employee`, `student :: PricingType`, které reprezentují typy zákazníků ze cvičení 4.1.10.

Příklady volání a správných vyhodnocení:

```
computePrice common   28 commonPricing ⇝* 338
computePrice employee 28 commonPricing ⇝* 287.30002
computePrice student  28 commonPricing ⇝* 28
```

4.2 Konstruktor `Maybe`

Př. 4.2.1  

Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. S výrazem `a` pracujte jako s externě definovaným typem.

a) `Maybe (Just 2)`
b) `Maybe a`
c) `Just a`
d) `Just Just 2`
e) `Maybe Nothing`
f) `Just Nothing`
g) `Nothing 3`
h) `[Just 4, Just Nothing]`
i) `Just [Just 3]`
j) `Just (\x -> x ^ 2)`
k) `\b matters -> if b then Nothing else matters`
l) `Just`
m) `Just Just`
n) `Just Just Just`

Př. 4.2.2 >>=

Definujte funkci `divlist :: Integral a => [a] -> [a] -> [Maybe a]`, s využitím typového konstruktoru `Maybe`, která celočíselně podělí dva celočíselné seznamy „po složkách“, tedy například

```
divlist [12, 5, 7] [3, 0, 2] ⇝* [Just 4, Nothing, Just 3]
divlist [12, 5, 7] [3, 1, 2, 5] ⇝* [Just 4, Just 5, Just 3]
divlist [42, 42] [0] ⇝* [Nothing]
```

a ošetří případy dělení nulou.

> **Pan Fešák doporučuje:** Pokud si nejste jistí, co funkce `zip` přesně dělá, tak se podívejte do dokumentace.

Př. 4.2.3 Napište funkci `mayZip :: [a] -> [b] -> [(Maybe a, Maybe b)]`, která je analogií funkce `zip`. Rozdílem je, že výsledný seznam má délku rovnou delšímu ze vstupních seznamů. Chybějící hodnoty jsou nahrazeny hodnotami `Nothing`.

4.3 Rekurzivní datové typy

Př. 4.3.1 Uvažme následující rekurzivní datový typ:

```
data Nat = Zero | Succ Nat deriving Show
```

a) Jaké hodnoty má typ `Nat`?
b) Jaký význam má dovětek `deriving Show`?
c) Redefinujte způsob zobrazení hodnot typu `Nat`.
d) Nadefinujte funkci `natToInt :: Nat -> Int`, která převede výraz typu `Nat` na číslo, které vyjadřuje počet použití hodnotového konstruktoru `Succ` v daném výrazu.
e) Jak byste pomocí datového typu `Nat` zapsali nekonečno?

Př. 4.3.2 Uvažme následující definici typu `Expr`:

```
data Expr = Con Float
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
```

a) Uveďte výraz typu `Expr`, který představuje hodnotu 3.14.
b) Definujte funkci `eval :: Expr -> Float`, která vrátí hodnotu daného výrazu.
c) Ošetřete korektně dělení nulou pomocí funkce `evaluate :: Expr -> Maybe Float`.

Př. 4.3.3 Rozšiřte definici z předchozího příkladu o nulární hodnotový konstruktor `Var`, který bude zastupovat proměnnou. Funkci `eval` upravte tak, aby jako první argument vzala hodnotu proměnné a vyhodnotila výraz z druhého argumentu pro dané ohodnocení proměnné.

*
**

> **Paní Bílá vysvětluje:** Binární strom (`BinTree a`) je struktura, která v každém svém uzlu `Node` udržuje hodnotu typu `a` a ukazatele na své dva potomky. Hodnotový konstruktor `Empty` nenese žádnou hodnotu a reprezentuje prázdný uzel bez potomků.

V následujících příkladech se využívá datová struktura

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

Př. 4.3.4

a) Nakreslete všechny tříuzlové stromy typu `BinTree ()` a zapište je pomocí hodnotových konstruktorů `Node` a `Empty`.
b) Kolik existuje stromů typu `BinTree ()` s 0, 1, 2, 3, 4 nebo 5 uzly?
c) Kolik existuje stromů typu `BinTree Bool` s 0, 1, 2, 3, 4 nebo 5 uzly?

> **Pan Fešák doporučuje:** Pro testování funkcí pracujících se strukturou `BinTree` můžete použít stromy, které najdete v souboru 📎 `04_trees.hs` v příloze sbírky nebo ve studijních materiálech v ISu.

Př. 4.3.5 »=

Pro datový typ `BinTree a` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu. Definujte následující funkce nad binárními stromy:

a) `treeSize :: BinTree a -> Int`, která spočítá počet uzlů ve stromě.

b) `listTree :: BinTree a -> [a]`, která vrátí seznam hodnot, které jsou uložené v uzlech vstupního stromu (na pořadí nezáleží),

c) `height :: BinTree a -> Int`, která určí výšku stromu.

d) `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě začínající v kořeni a vrátí ohodnocení na ní.

Pár příkladů vyhodnocení funkcí v této úloze:

```
treeSize Empty ⇝* 0
treeSize tree01 ⇝* 7
listTree tree01 ⇝* [1, 2, 3, 4, 5, 6, 7] -- Jedno z možných řešení.
listTree tree02 ⇝* [9, 10, 11, 12]
height tree02 ⇝* 4 longestPath tree05 ⇝* [100, 101, 102, 103, 104]
```

Př. 4.3.6

Pro datový typ `BinTree a` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.

a) Definujte funkci `fullTree :: Int -> a -> BinTree a`, která pro volání `fullTree n v` vytvoří binární strom výšky `n`, ve kterém jsou všechny větve stejně dlouhé a všechny uzly jsou ohodnocené hodnotou `v`.

b) Definujte funkci `treeZip :: BinTree a -> BinTree b -> BinTree (a, b)` jako analogii seznamové funkce `zip`. Výsledný strom tedy obsahuje pouze ty uzly, které jsou v obou vstupních stromech.

Př. 4.3.7

Napište `treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)` jako analogii seznamové funkce `mayZip` z příkladu 4.2.3. Vrchol v novém stromu bude existovat právě tehdy, pokud existuje aspoň v jednom ze vstupních stromů.

Př. 4.3.8 »=

Deklarujte typ `BinTree a` jako instanci typové třídy `Eq`. Instanci si napište sami (tj. nepoužívejte klauzuli `deriving`).

Př. 4.3.9

Uvažme datový typ `BinTree a`.

a) Definujte funkci `isTreeBST :: (Ord a) => BinTree a -> Bool`, která se vyhodnotí na `True`, jestli bude její první argument validní binární vyhledávací strom.

b) Definujte funkci `searchBST :: (Ord a) => a -> BinTree a -> Bool`, která projde BST z druhého argumentu v smyslu binárního vyhledávání a vyhodnotí se na `True` v případě, že její první argument najde v uzlech při vyhledávání.

Můžete předpokládat, že vstupní datový typ `a` je uspořádaný lineárně.

4.4 Další příklady

Př. 4.4.1 Uvažte typ stromů s vrcholy libovolné arity definovaný následovně:

```
data RoseTree a = RoseNode a [RoseTree a]
                deriving (Show, Read)
```

Definujte následující:

a) funkci `roseTreeSize :: RoseTree a -> Int`, která spočítá počet uzlů ve stromě,

b) funkci `roseTreeSum :: Num a => RoseTree a -> a`, která sečte ohodnocení všech uzlů stromu,

c) funkci `roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b`, která bere funkci a strom a aplikuje danou funkci na hodnotu v každém uzlu:

```
roseTreeMap (+1) (RoseNode 0 [RoseNode 1 [], RoseNode 41 []])
    ⇝* RoseNode 1 [RoseNode 2 [], RoseNode 42 []]
```

Př. 4.4.2 Uvažujme rekurzivní datový typ `IntSet` definovaný takto:

```
data IntSet = SetNode Bool IntSet IntSet | SetLeaf  -- Node end zero one
            deriving Show
```

Ve stromě typu `IntSet` každá cesta z vrcholu jednoznačně určuje binární kód složený z čísel přechodů mezi otcem a synem (podle označení syna `one` respektive `zero`). Toho můžeme využít pro ukládání přirozených čísel do takového stromu. Strom typu `IntSet` obsahuje číslo n právě tehdy, pokud obsahuje cestu odpovídající binárnímu zápisu čísla n, a navíc poslední vrchol této cesty má nastavenou hodnotu `end` na `True`.

Implementujte tyto funkce pro práci se strukturou `IntSet`:

a) `insert :: IntSet -> Int -> IntSet` – obdrží strom typu `IntSet` a přirozené číslo n a navrátí strom obsahující číslo n.

b) `find :: IntSet -> Int -> Bool` – obdrží strom typu `IntSet` a přirozené číslo n a vrátí `True` právě tehdy, pokud strom obsahuje n.

c) `listSet :: IntSet -> [Int]` – obdrží strom typu `IntSet` a navrátí seznam čísel uložených v tomto stromě.

Př. 4.4.3 Podobná stromová struktura jako v příkladu 4.4.2 by mohla být použita i pro udržování množiny řetězců nad libovolnou abecedou (například slova složená z písmen anglické abecedy nebo konečné posloupnosti celých čísel). Definujte datový typ `SeqSet a` sloužící pro uchovávání posloupností prvků typu `a`. Dále definujte obdoby funkcí ze cvičení 4.4.2:

a) `insertSeq :: Eq a => SeqSet a -> [a] -> SeqSet a`

b) `findSeq :: Eq a => SeqSet a -> [a] -> Bool`

*
**

Na konci cvičení byste měli zvládnout:

- tvorbu vlastních datových typů,
- umět implementovat jednoduché instance typových tříd pro vlastní datový typ,
- využívat datový typ `Maybe`,
- implementovat funkce na rekurzivních datových typech, a to především na strukturách typu strom.

Cvičení 5: Lenost, intensionální seznamy, foldy

Před pátým cvičením je zapotřebí znát:

- co je to vyhodnocovací strategie;
- jak probíhá striktní a líné vyhodnocování;
- co jsou intensionální seznamy a jak se v Haskellu zapisují, tj. například umět přečíst zápis

```
[ 2 * y | x <- [1,2,3,4,5], even x, let y = 2 + x ]
```

- 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
```

5.1 Lenost

Př. 5.1.1 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.

Př. 5.1.2 Uvažte seznam `naturals`, který jste definovali v předchozím příkladu. 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. 5.1.3 Uvažte opět seznam `naturals` z příkladu 5.1.1. Mějme dále standardní 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.

Př. 5.1.4 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. 5.1.5 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`).

Př. 5.1.6 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. 5.1.7 Jaké jsou výhody líné vyhodnocovací strategie? Jaké jsou výhody striktní vyhodnocovací strategie?

Př. 5.1.8 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. 5.1.9 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. 5.1.10 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. 5.1.11 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. 5.1.12 Pomocí rekurzivní definice a funkce `zipWith` vyjádřete Fibonacciho posloupnost tak, že pro výpočet každého prvku posloupnosti stačí lineárně mnoho rekurzivních volání.

Př. 5.1.13 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. 5.1.14 Definujte nějaký binární strom, který obsahuje alespoň jednu nekonečnou větev a alespoň jednu konečnou větev.

5.2 Intensionální seznamy

Př. 5.2.1 Reprezentujme seznam studentů a jimi absolvovaných předmětů pomocí typu `[(Int, [String])]`, kde první složka dvojice reprezentuje učo studenta a druhá složka kódy předmětů, které daný student absolvoval.

Pomocí intensionálních seznamů napište funkce:

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

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

c) `passedIB015 :: [(Int, [String])] -> [Int]`, která vrátí uča studentů, kteří absolvovali předmět `"IB015"`,
d) `passedBySomeone :: [(Int, [String])] -> [String]`, která vrátí kódy předmětů, které absolvoval alespoň jeden student (kódy se v seznamu můžou opakovat).

Př. 5.2.2 Reprezentujme seznam účastníků plesu pomocí typu `[(String, Bool)]`, kde první složka dvojice reprezentuje jméno účastníka a druhá složka je `True` právě tehdy, když účastník je žena.

Pomocí intensionálních seznamů napište funkci

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

která pro daný seznam účastníků vrátí seznam všech možných dvojic účastníků ve tvaru `(muz, zena)`. Například tedy:

```
allPairs [("Jeff", False), ("Britta", True),
          ("Annie", True), ("Troy", False)] ⇝*
    [("Jeff", "Britta"), ("Jeff", "Annie"),
     ("Troy", "Britta"), ("Troy", "Annie")]
```

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

Př. 5.2.3 Pomocí intensionálních seznamů definujte funkci `divisors`, která k zadanému přirozenému číslu vrátí seznam jeho kladných dělitelů.

Př. 5.2.4 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. 5.2.5 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. 5.2.6 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. 5.2.7 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 _ = []
  ```

Př. 5.2.8 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. 5.2.9 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. 5.2.10 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. 5.2.11 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

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 Pomocí vhodné akumulační funkce `foldr`, `foldl`, `foldr1` nebo `foldl1` implementujte následující funkce:

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 `andFold`, která vrátí `False`, pokud se v zadaném seznamu nachází aspoň jednou hodnota `False`, jinak vrátí `True`.
```

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

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

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

f) Funkci `minimumFold`, která vrátí minimální prvek ze zadaného neprázdného seznamu.

```
minimumFold :: Ord a => [a] -> a
minimumFold [3, 4, 2, 5] ⇝* 2
```

g) 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 Určete, co dělají akumulační funkce s uvedenými argumenty. Najděte hodnoty, na které lze tyto výrazy aplikovat, a ověřte pomocí interpretu.

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

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 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.

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

a) 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
```

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

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

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

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

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

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

e) 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]
```

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

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

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 minimální a maximální 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!

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

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] ⇝* [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 `foldr f [] s` vyhodnotilo na seznam obsahující jenom prvky ze sudých míst v seznamu `s`?

Je možné definovat takovou funkci pomocí výrazu ve tvaru `snd (foldr 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 strukturách

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 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 

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 volání `roseTreeFold n e t` nahradí ve stromě `t` všechny hodnotové konstruktory `RoseNode` funkcí `n` a všechny hodnotové konstruktory `RoseEmpty` hodnotou `e`. 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`.

*
**

Na konci pátého cvičení byste měli umět:

- použít líné vyhodnocování k práci s nekonečnými seznamy;
- umět použít nekonečné seznamy v praxi;
- 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 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.

Cvičení 6: Vstup a výstup

Před šestým cvičením je zapotřebí:

- znát význam pojmů *vstupně-výstupní akce* a *vnitřní výsledek*;
- rozumět, co znamenají typy jako `IO Integer` a `IO ()`, a chápat jejich odlišnost od `Integer` a `()`;
- vědět, k čemu slouží funkce:

```
pure      :: a -> IO a
getLine   :: IO String
putStrLn  :: String -> IO ()
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

- být seznámeni s operátory pro skládání vstupně-výstupních akcí:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>)  :: IO a -> IO b -> IO b
```

- být seznámeni se syntaxí a významem konstrukce `do`:

```
main = do input <- getLine
          let hello = "Hello there, "
          putStrLn (hello ++ input)
```

- umět přeložit zdrojový kód na spustitelný program pomocí `ghc` ***`zdroják`***.

Vstup a výstup v Haskellu je na první pohled zvláštní z prostého důvodu – vyhodnocení výrazu nemůže mít vedlejší efekty, jako je výpis na obrazovku. Je ale možné výraz vyhodnotit na „scénář“ obsahující popis vstupně-výstupních operací: kdy se má načítat vstup, co se má vkládat do kterých souborů apod. Při samotném vyhodnocení se sice nic nestane, ale podle výsledného scénáře pak může vstup a výstup provádět třeba interpret.

Těmto scénářům říkáme *vstupně-výstupní akce*. Hodnota typu `IO a` je pak scénář pro získání `a`. Zde `a` je typ *vnitřního výsledku* akce, který je naplněn po vykonání akce. Vnitřní výsledek (například načtený řetězec) nemůže vstupovat do vyhodnocování výrazů, ale jiné akce jej používat mohou (například ho vypsat na obrazovku).

K takovému řetězení scénářů do větších akcí slouží operátor `>>=`. Povšimněte si jeho typu: levý argument je „scénář pro získání `a`“, pravý zjednodušeně dává „scénář, podle kterého se z `a` vyrobí `b`[3]“.

Akce jako hodnoty jsou pro nás zcela neprůhledné – nemáme možnost zjistit, jaké efekty bude akce mít, aniž ji spustíme (a tedy všechny efekty provedeme). *Spouštění* je kromě spojování přes `>>=` v podstatě jediná zajímavá věc, kterou s akcemi můžeme dělat. Spouští se každá akce, kterou si necháme vyhodnotit v interpretu, a v samostatných spustitelných souborech akce `main :: IO ()`.

[3] Ve skutečnosti spíše „jak z `a` vyrobit scénář pro `b`“. Pro zkrácení ale můžeme např. funkci typu `String -> IO Integer` označovat jako „akci, která bere řetězec a vrací číslo“, byť formálně správně se jedná o „funkci, která bere řetězec a vrací vstupně-výstupní akci s vnitřním výsledkem typu číslo“.

> **Jindřiška varuje:** Vstup a výstup je matoucí, pokud důsledně nerozlišujeme mezi pojmy *hodnota*, *(vstupně-výstupní) akce* a *vnitřní výsledek akce*. Plést si nesmíme ani *vyhodnocení* a *spuštění*.

6.1 Skládání akcí operátorem >>=

Př. 6.1.1 Pomocí známých funkcí a operátoru `>>=` definujte vstupně-výstupní akci, která po spuštění přečte řádek ze standardního vstupu a:

a) vypíše jej beze změny na standardní výstup;
b) vypíše jej pozpátku;
c) vypíše jej, není-li prázdný, jinak vypíše „`<empty>`“;
d) vypíše jej a také jej uloží jako vnitřní výsledek akce (bude se hodit i operátor `>>`).

Př. 6.1.2 Bez použití `do`-notace definujte akci `getInteger :: IO Integer`, která ze standardního vstupu načte celé číslo. Využijte knihovní funkci `read :: (Read a) => String -> a`.

Př. 6.1.3 Bez použití `do`-notace definujte vstupně-výstupní akci `loopecho :: IO ()`, která při spuštění načítá a vypisuje řádky do té doby, než načte prázdný řádek.

Př. 6.1.4 Napište akci `getSanitized :: IO String`, která načte jeden řádek textu od uživatele a z něj odstraní všechny znaky, které nejsou znaky abecedy. V úkolu použijte funkci `isAlpha` z modulu `Data.Char`.

Př. 6.1.5 Bez použití `do`-notace definujte akci, která ze standardního vstupu přečte cestu k souboru a následně uživateli oznámí, zda zadaný soubor existuje. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`.

Př. 6.1.6 Definujte funkci `(>>)` pomocí funkce `(>>=)`.

Př. 6.1.7 Napište funkce `runLeft :: [IO a] -> IO ()` a `runRight :: [IO a] -> IO ()`, které spustí všechny akce v zadaném seznamu postupně zleva (respektive zprava).

6.2 IO pomocí do-notace, převody mezi notacemi

do-notace	***bind*-notace**
`do f` `  g`	`f >> g`
`do x <- f` `  g`	`f >>= \x -> g`
`do let x = y` `  f`	`let x = y in f`

Př. 6.2.1 Vraťte se k některému ze svých řešení příkladů z předchozí sekce a přepište je na `do`-notaci. Pro ilustraci zkuste přepsat jedno snadné a jedno mírně složitější řešení a srovnejte je s řešením pomocí `>>=`.

Př. 6.2.2 Naprogramujte funkci `leftPadTwo :: IO ()`, která od uživatele načte dva řetězce a pak je vypíše na obrazovku zarovnány doprava tak, že před kratší z nich vypíše ještě vhodný počet mezer. Použijte notaci `do`.

Př. 6.2.3 Převeďte následující program v `do`-notaci na notaci s použitím `>>=`.

```
main = do
    f <- getLine
    s <- getLine
    appendFile f (s ++ "\n")
```

Př. 6.2.4 Následující funkci přepište do tvaru, ve kterém nepoužijete konstrukci `do`. Určete také typ funkce.

```
query question = do putStrLn question
                    answer <- getLine
                    pure (answer == "ano")
```

Př. 6.2.5 Funkci `query` z předchozího příkladu dále vylepšete tak, aby:

a) Rozlišovala kladné i záporné odpovědi a při nekorektní nebo nerozpoznané odpovědi otázku opakovala.

b) Akceptovala odpovědi s malými i velkými písmeny, interpunkcí, případně ve více jazycích.

Př. 6.2.6 U každého z následujících výrazů určete typ a význam, případně vysvětlete, proč není korektní:

a) `getLine`

b) `x = getLine`

c) `let x = getLine in x`

d) `let x <- getLine in x`

e) `getLine >>= \x -> pure x`

f) `getLine >>= \x -> x`

g)
```
do let x = getLine
   pure x
```

h)
```
do let x = getLine
   x
```

i)
```
do x <- getLine
   pure x
```

j)
```
do x <- getLine
   x
```

6.3 Vstupně-výstupní programy

Př. 6.3.1 Vyrobte a spusťte program, který se při chová jako akce `leftPadTwo` z příkladu 6.2.2. Připomínáme, že zdrojový kód se na spustitelný soubor překládá programem `ghc` a musí mít zadefinovanou akci `main :: IO ()`. Výsledný program se jmenuje jako zdrojový soubor bez přípony `.hs` a je nutné ho spouštět pomocí *`./program`*.

Př. 6.3.2 Upravte a doplňte následující zdrojový kód tak, aby program vyžadoval a načetl postupně tři celá čísla a o nich určil, zda mohou být délkami hran trojúhelníku. Akce `getInteger` pochází z příkladu 6.1.2. Nezdráhejte se kód refaktorovat a vytvořit si další pomocné funkce či akce.

```
main :: IO ()
main = do putStrLn "Enter one number:"
          x <- getInteger
          putStrLn (show (1 + x))
```

Př. 6.3.3 Napište program, který vyzve uživatele, aby zadal jméno souboru, a poté ověří, že zadaný soubor existuje. Pokud existuje, vypíše jeho obsah na obrazovku, pokud ne, informuje o tom uživatele. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`.

Př. 6.3.4 Vysvětlete význam a rizika rekurzivního použití akce `main` v následujícím programu.

```
main :: IO ()
main = do putStr "Enter string: "
          s <- getLine
          if null s then putStrLn "You shall not pass!"
          else do putStrLn (reverse s)
          main
```

Př. 6.3.5 V dokumentaci nalezněte vhodnou funkci typu `Read a => String -> Maybe a` a s jejím využitím napište funkci `requestInteger :: String -> IO (Maybe Integer)` použitelnou na chytřejší načítání čísel. Akce `requestInteger` *`delim`* od uživatele čte řádky tak dlouho, než dostane řetězec *`delim`* (potom vrátí `Nothing`), nebo celé číslo (které vrátí zabalené v `Just`). Po každém neúspěšném pokusu by měl uživatel dostat výzvu k opětovnému zadání čísla.

Př. 6.3.6 S využitím akce z předchozího příkladu napište program, který která ze standardního vstupu čte řádky s čísly, dokud nenarazí na prázdný řádek, potom vypíše jejich aritmetický průměr. Vyřešte úlohu:

a) s ukládáním čísel do seznamu;
b) bez použití seznamu.

Př. 6.3.7 Napište program `leftPad`, který je obecnější variantou akce z příkladu 6.2.2 a zarovnává libovolný počet řádků na vstupu, dokud nenalezne řádek obsahující jen tečku.[4] Řádky jsou opět zarovnány doprava na délku nejdelšího.

Následně proveďte drobnou optimalizaci: prázdné řádky nechte prázdnými, tj. bez zbytečných výplňových mezer.

Př. 6.3.8 Upravte program `06_guess.hs` tak, aby parametry funkce `guess` četl z příkazové řádky. Vhod může přijít modul `System.Environment`.

Př. 6.3.9 Vymyslete a naprogramujte několik triviálních prográmků manipulujících s textovými soubory:

- počítání řádků
- výpis konkrétního řádku podle zadaného indexu
- vypsání obsahu pozpátku
- seřazení řádků, ...

Definice alternativně přepište s a bez pomoci syntaktické konstrukce `do`.

Př. 6.3.10 Představme si bohy zatracený svět, v němž neexistuje konstruktor `IO` ani vstupně-výstupní akce, které je nutné spouštět. Místo toho k vedlejším efektům dochází rovnou při vyhodnocování výrazů. Např. `getLine' :: String` se *vyhodnotí* na řádek vstupu. Vyhodnocovací strategie ale funguje stejně, jak ji v Haskellu známe. Co bude výsledkem úplného vyhodnocení výrazu `(getLine', getLine')`?

*
**

[4] Proč zrovna tečku, ptáte se? Bylo nebylo... https://en.wikipedia.org/wiki/ed_(text_editor), https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol#SMTP_transport_example

Na konci cvičení byste měli zvládnout:

- napsat v Haskellu jednoduchý program pracující se vstupem od uživatele a vypisující informace na výstup;
- za pomoci dokumentace napsat jednoduchý program pracující se soubory;
- převádět mezi konstrukcí `do` a operátory `>>=` a `>>`.

Zdá se, že vstup a výstup vyžaduje velmi odlišný přístup, speciální operátory a syntaxi a celé to může působit velmi nehaskellovsky. Ve skutečnosti jsou ale podobné konstrukce Haskellu vlastní a běžně se používají i mimo `IO`! Napovídat tomu může typ operátoru `>>=`, který zmiňuje jakousi `Monad`. *Monáda* je velmi abstraktní, avšak fascinující koncept, o němž se můžete více dozvědět v jarním semestru v navazujících předmětech IB016 *Seminář z funkcionálního programování* a IA014 *Advanced Functional Programming*.

Řešení

Cvičení 0: Technické okénko

Řeš. 0.0.0 Gratulki. Klepnutím na číslo řešení přeskočíte zpátky na příklad.

Řeš. 0.1.2 Vytvořit jej můžete buď v terminálu (příkazem `mkdir` *`jméno`* a následně do něj přepnout pomocí `cd` *`jméno`*), nebo v grafickém správci souborů, který by se měl otevřít v domovském adresáři a nový adresář by mělo být lze vytvořit pravým myšítkem nebo přes panel nabídek.

Řeš. 0.1.3 Spusťte textový editor (například Gedit) a napište či překopírujte do něj text. Nezapomeňte soubor uložit do správného adresáře.

Alternativně můžete soubor vytvořit i v terminálu, například editorem `nano`, který vám v dolní části zobrazuje klávesové zkratky, kterými se ovládá (`^` znamená Ctrl).

Jiným oblíbeným terminálovým editorem je `vim`. Jeho obsluha vypadá zpočátku krkolomně, ale dá se rychle naučit: zkuste příkaz `vimtutor` či `vimtutor sk`.

Řeš. 0.2.2 Zadejte výraz a stiskněte Enter. Např. `40 + 2` se vyhodnotí na `42`. Vyhodnocení výrazů zapisujeme ve studijních materiálech pomocí lomené šipky s hvězdičkou: `40 + 2` $\rightsquigarrow^*$ `42`. Další příklady:

- `4 * 10 + 2` $\rightsquigarrow^*$ `42`
- `2 ^ 8` $\rightsquigarrow^*$ `256`
- `(3 + 4) * 6` $\rightsquigarrow^*$ `42`

Řeš. 0.2.3 Načíst soubor lze buď pomocí `:load Sem0.hs` nebo jen `:l Sem0.hs` (pokud se nachází ve stejném adresáři, kde jsme spustili GHCi), nebo tak, že spustíme GHCi přímo s tímto názvem souboru (`ghci Sem0.hs`). Konstantu vypíšeme prostě tak, že ji napíšeme do GHCi a stiskneme enter: `hello` $\rightsquigarrow^*$ `"Hello, world"`. Její typ pak zjistíme příkazem interpretu `:t`, který jako parametr bere libovolný výraz: `:t hello` $\rightsquigarrow^*$ `hello :: [Char]`.

Řeš. 0.2.4

```
my_number :: Integer
my_number = 42
```

Znovunačtení lze provést příkazem `:r`, ukončení `:q`.

Řeš. 0.3.1 Na Linuxu, macOS a Windows s dostatečně novým PowerShellem by mělo stačit zadat do příkazové řádky (či PowerShellu) `ssh xLOGIN@aisa.fi.muni.cz` a následně zadat své heslo; tím se ocitnete na Aise a můžete tam používat příkazovou řádku jako na cvičení. Na Windows s Putty musíte nastavit sezení (*session*), kde jako hostitele uvedete `xLOGIN@aisa.fi.muni.cz`.

Pro trvalé přidání modulu můžete příkaz vložit do svého souboru `.bashrc`:

```
$ echo "module add ghc" >> ~/.bashrc  # Nebo textovým editorem
```

Řeš. 0.3.2

```
$ scp xLOGIN@aisa.fi.muni.cz:ib015/Sem0.hs .
$ vim Sem0.hs       # Nebo jiný způsob editace
$ scp Sem0.hs xLOGIN@aisa.fi.muni.cz:ib015/Sem0_edited.hs
```

Poznámka: Znakem `$` se běžně uvozuje text zadaný do terminálu, není ale součástí zadávaného příkazu.

Řeš. 0.3.3 Do souboru `~/.ssh/config` na svém počítači vložte:

```
Host aisa
    HostName %h.fi.muni.cz
```

```
User xLOGIN
```

Cvičení 1: Základní konstrukce

Řeš. 1.1.1 Jděte na `https://hoogle.haskell.org/`. V rozbalovací nabídce u vyhledávacího políčka zvolte `package:base` a do vyhledávacího políčka zadejte hledaný typ. Po chvíli by vám vyhledávač měl najít funkce `(&&)`, `(||)`, `(==)` a `(/=)` (kde poslední dvě mají obecnější typ, ale fungují i pro argumenty typu `Bool`). U každé funkce také vidíte, ve kterém balíčku (`base`) a modulu (`Prelude`[5]) se nachází, a také dokumentační text, pokud je uveden. Kliknutím na funkci se dostanete do její dokumentace v prvním modulu, kde byla nalezena.

Řeš. 1.2.1 Celkově zjistíme, že operátory jsou uvedeny v zadání v pořadí od nejvyšší priority (9) až k nejnižší (1).

Poznámka: Ve skutečnosti existují i operátory s prioritou 0, například `$`, ke kterému se časem dostaneme.

Řeš. 1.2.2 a) V důsledku priorit operátorů je implicitní závorkování kolem násobení, tj. `5 + (9 * 3)`.

b) Operace umocňování asociuje zprava, tedy v případě více výskytů `(^)` za sebou se implicitně závorkuje zprava. Obecně tedy

`n ^ n ^ n`	=	`n ^ (n ^ n)`	≠	`(n ^ n)^ n`	=	`n ^ (n ^ 2)`
n^{n^n}	=	$n^{(n^n)}$	≠	$(n^n)^n$	=	$n^{(n^2)}$

c) Tady se setkáváme s případem, kdy je operátor neasociativní, tedy není definováno, jak se výraz zpracovává v případě výskytu více operátorů stejné priority vedle sebe, a výraz je tedy nekorektní. Důvod neasociativity je jednoduchý: u `(==)` totiž nemá smysl definovat asociativitu, protože jeho výsledek je typu `Bool`, ale argumenty mohou být jiného typu – to nakonec vidíme i na našem příkladě `3 == 3 == 3`, ať jej uzávorkujeme libovolně, nebudou nám sedět typy (budeme porovnávat číslo a `Bool`).

d) V případě, že explicitně uvedeme závorkování pro relační operátory, dostaneme se do obdobné situace jako v předchozím podpříkladu, tedy porovnání logické hodnoty a řetězce, což v Haskellu nelze. Naproti tomu výsledkem porovnání `'a' == 'a'` dostaneme v obou případech logickou hodnotu, a ty mezi sebou porovnávat můžeme, protože jsou stejného typu. Všimněte si, že v tomto výrazu se vyskytuje `==` jednou ve verzi pro znaky (`'a' == 'a'`) a jednou pro `Bool` (prostřední výskyt).

Řeš. 1.2.3 Je důležité zachovávat pořadí operandů! I když jde o komutativní operátor, nelze obecně při změně mezi prefixovým a infixovým zápisem měnit jejich pořadí, protože vzniklý výraz nebude totožný. Navíc Haskell nijak negarantuje, že např. `+` je komutativní.

a) `(^) 4 (mod 7 5)`
b) `3 `max` (2 + 3)`

Řeš. 1.2.4 Při doplňování implicitních závorek je potřeba se řídit prioritou/asociativitou infixově zapsaných operátorů a závorkováním aplikace funkcí na argumenty. Postupujeme následovně:

1. Obsahuje-li výraz infixově zapsané operátory, najdeme ty s nejnižší prioritou, které nejsou v závorkách, a jejich operandy uzávorkujeme. Pokud je těchto operátorů více než jeden, jednotlivé operandy závorkujeme dle asociativity daných operátorů (pokud se vedle sebe vyskytují dva operátory se stejnou prioritou, ale různou asociativitou nebo bez asociativity, výraz je nesprávně utvořený).

[5] `Prelude` je základní Haskellový modul, který je vždy k dispozici.

2. Pokud již ve výrazu nejsou infixové operátory, tj. výraz je jednoduchý (konstanta, proměnná nebo název funkce), prefixová aplikace funkce nebo aplikace infixově zapsaného binárního operátoru na dva jednoduché argumenty, skončili jsme.

3. V opačném případě aplikujeme stejný postup na všechny podvýrazy vzniklé buď doplněnými závorkami, nebo dosud nezpracovanými závorkami v původním výrazu.

Řešení jednotlivých příkladů budou následující:

a) `(recip 2) * 5`
b) `(sin pi) / 2`
c) `(mod 3 8) * 2`
d) `(f g 3) + (g 5)` (funkce `f` je aplikována na 2 argumenty)
e) `(42 < 69) || (5 == 6)`
f) `((2 + (div m 18)) == (m ^ (2 ^ n))) && ((m * n) < 20)`

Řeš. 1.2.5

a) `f . (g x)`
b) `2 ^ (mod 9 5)`
c) `f . (((.) g h) . id)`
d) `((2 + (((div m 18) * m) `mod` 7)) == (((m ^ (2 ^ n)) - m) + 11)) && ((m * n) < 20)`
e) `(f 1 2 g) + (((+) 3) `const` (g f 10))`
f) `(replicate 8 x) ++ ((filter even) (enumFromTo 1 (3 + (9 `mod` x))))`
g) `(id id) . (flip const const)`

Řeš. 1.2.6 Potřebujeme zjistit, jestli se `y` rovná číslu o jedna většímu než `x`.

```
isSucc :: Integer -> Integer -> Bool
isSucc x y = y == x + 1
```

Řeš. 1.2.7 Obsah kruhu o poloměru r se vypočítá vzorečkem πr^2. Do Haskellu to snadno přepíšeme jako:

```
circleArea :: Double -> Double
circleArea r = pi * r ^ 2
```

Řeš. 1.2.8

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z = max x (max y z)

max3' :: Integer -> Integer -> Integer -> Integer
max3' x y z = if x > y
    then if z > x then z else x
    else if z > y then z else y
```

Alternativní řešení pomocí konstrukce `if ... then ... else ...`:

```
max3'' :: Integer -> Integer -> Integer -> Integer
max3'' x y z = if x > y && x > z
    then x
    else if z > y then z else y
```

Řeš. 1.2.9 Nejprve zjistíme, která strana je nejdelší, a pak strany pošleme ve správném pořadí do rovnice pro Pythagorovu větu, kterou si zadefinujeme pomocí lokální definice.

```
isRightTriangle :: Integer -> Integer -> Integer -> Bool
isRightTriangle x y z = if x >= y && x >= z
    then pyt y z x      -- x is the maximum
    else if y >= z      -- x is not the maximum, one of y or z must be
         then pyt x z y -- y is the maximum
         else pyt x y z -- z is the maximum
    where
        pyt a b c = a ^ 2 + b ^ 2 == c ^ 2
```

Alternativou je vyzkoušet všechny tři možné volby pro nejdelší stranu (víme jistě, že pokud vybereme některou z kratších stran místo té nejdelší, rovnost v Pythagorově větě nevyjde). I zde můžeme s výhodou využít lokální definice.

```
isRightTriangle' :: Integer -> Integer -> Integer -> Bool
isRightTriangle' x y z = pyt x y z || pyt y z x || pyt x z y
    where
        pyt a b c = a ^ 2 + b ^ 2 == c ^ 2
```

Řeš. 1.2.10 Přímočaré je řešení pomocí `if` a `min`/`max`, stačí si uvědomit, že prostřední číslo je menší nebo rovno maximu a větší nebo rovno minimu:

```
mid :: Integer -> Integer -> Integer -> Integer
mid x y z = if min y z <= x && x <= max y z
    then x
    else if min x z <= y && y <= max x z
        then y
        else z
```

Alternativou je vypočítat prostřední číslo za pomoci součtu, minima a maxima:

```
mid' :: Integer -> Integer -> Integer -> Integer
mid' x y z = x + y + z - max z (max x y) - min z (min x y)
```

Další alternativou je využít sílu funkce `max3`, kterou jsme již definovali (předpokládejme tedy, že je definována). Ta nám umožňuje vybírat to největší z libovolných tří celých čísel. Pokud tedy vybereme čísla z původní trojice tak, že máme zaručeno, že vybereme všechna kromě toho největšího (některé se může zopakovat), maximem z těchto čísel bude právě prostřední prvek v uspořádání podle $\leq$:

```
mid'' :: Integer -> Integer -> Integer -> Integer
mid'' x y z = max3 (min x y) (min y z) (min x z)
```

Řeš. 1.2.11 Řešení je zdlouhavé, ale celkem přímočaré. Pokud víme, že číslo n je sudé, právě když $n \bmod 2 = 0$, stačí jenom zbylé podmínky vypsat do zanořených `if`ů a dát pozor na to, že `if` v Haskellu má vždy i neúspěšnou větev po `else`. Zároveň obě větve musí být stejného *typu* (co intuitivně znamená, že obě musí vracet číslo nebo znak nebo řetězec…):

```
tell :: Integer -> String
tell n = if n > 2
         then if mod n 2 == 0 then "(even)" else "(odd)"
         else if n == 1 then "one" else "two"
```

Vnořené `if`y lze i uzávorkovat, ale je to zbytečné.

Toto řešení je poněkud špatně čitelné a použití `if` v Haskellu není vždy žádoucí. Časem si ukážeme něco lepšího.

Řeš. 1.2.12 a) Podmínka musí být logický výraz typu `Bool`, což výraz `5 - 4` není – jde o výraz celočíselného typu. Haskell nikdy sám nekonvertuje výrazy jednoho typu na druhý. Vhodná úprava celého výrazu je pak třeba `5 - 4 == 0`.

b) Výrazy v `then` a `else` větvi musí být stejného typu, protože celý podmínkový výraz musí mít vždy stejný typ bez ohledu na hodnotu podmínky. Výraz lze opravit na

```
if 0 < 3 && odd 6 then "OK" else "FAIL"
```

což už je typově správně.

c) Na první pohled podivně vypadající konstrukce, kde výsledkem podmínkového výrazu je operátor `(&&)`, je správná. V Haskellu jsou funkce/operátory rovnocenné s číselnými či jinými konstantami. Problémem je chybějící větev `else`. Podmíněný výraz má syntaktické omezení, že vždy musí obsahovat jak `then`, tak `else` větev, i když by podmínka zaručovala použití jen jedné z nich. Kdyby podmínka mohla být vyhodnocena na nepravdu a chybělo by `else`, pak by výraz neměl žádnou hodnotu, kterou by vrátil. Ale výraz v Haskellu vždy musí mít nějakou hodnotu a je tedy třeba přidat `else` větev:

```
if even 8 then (&&) else (||)
```

d) Tento výraz je v pořádku. A to i přes to, že v interpretu dostaneme podivnou hlášku:

```
> if 42 < 42 then (&&) else (||)

<interactive>:1:1: error:
    • No instance for (Show (Bool -> Bool -> Bool))
        arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
```

Tato hláška však jen říká, že výslednou hodnotu výrazu nelze vypsat (kritická je tu ta část „`In a stmt of an interactive GHCi command: print it`“, která říká, že se jedná o chybu při vypisování výsledku výrazu). Konkrétně zde se interpret snaží vypsat hodnotu typu `Bool -> Bool -> Bool`, tedy binární funkci, která bere dvě pravdivostní hodnoty a jednu vrací. Funkce ale nelze v GHCi za normálních okolností vypisovat. Pokud nebudete chtít výsledek výrazu vypsat, ale jen ho otypujete pomocí příkazu `:t if 42 < 42 then (&&) else (||)`, k žádné chybě nedojde.

Řeš. 1.2.13 Nejdříve podle priority operátorů do výrazu zapíšeme implicitní závorky (kvůli různým prioritám operátorů):

```
(5 + (((7 * 5) `mod` 3) `div` 2)) == ((3 * 2) - 1)
```

Pak už lehce zjistíme, že výraz se vyhodnotí na `False`.

Při vyhodnocování výrazu v zadání se jako poslední vyhodnotí funkce s nejnižší prioritou, v našem případě `(==)`. Přepíšeme tedy do prefixu nejdříve tuto funkci:

```
(==) (5 + 7 * 5 `mod` 3 `div` 2) (3 * 2 - 1)
```

Následně v každém z argumentů opět najdeme funkci s nejnižší prioritou – v prvním je to funkce `(+)`, ve druhém pak `(-)`. Přepisem těchto funkcí do prefixu dostaneme:

```
(==) ((+) 5 (7 * 5 `mod` 3 `div` 2)) ((-) (3 * 2) 1)
```

Stejným způsobem pokračujeme i nadále. Jestliže narazíme na skupinu operátorů se stejnou prioritou (například `(*), mod, div`), ověříme si jejich směr sdružování (závorkování). V našem případě se sdružuje (závorkuje) zleva. To v praxi znamená, že jako poslední

se vyhodnotí funkce `div`. Výraz tedy přepíšeme následovně:

```
div (7 * 5 `mod` 3) 2
```

Stejným způsobem pokračujeme, dokud nám nezůstanou žádné infixově zapsané operátory:

```
(==) ((+) 5 (div (mod ((*) 7 5) 3) 2)) ((-) ((*) 3 2) 1)
```

Řeš. 1.2.14 Se závorkami:

```
(((2 + (2 * 3)) == (2 * 4)) && (((8 `div` 2) * 2) == 2)) || (0 > 7)
```

A po přepisu do prefixu:

```
(||) ((&&) ((==) ((+) 2 ((*) 2 3)) ((*) 2 4))
           ((==) ((*) (div 8 2) 2) 2))
     ((>) 0 7)
```

Řeš. 1.3.1 Je potřeba se zamyslet nad tím, které případy má smysl vytáhnout pomocí vzorů jako ty „speciální“. V našem případě to budou právě víkendové dny a pro vše ostatní vrátíme `False`.

```
isWeekendDay "Saturday" = True
isWeekendDay "Sunday"   = True
isWeekendDay _          = False
```

Řeš. 1.3.2 Opět se zamyslíme nad tím, které případy vytáhnout do vzorů. V tomto případě to budou samohlásky, protože samohlásek je v anglické abecedě oproti souhláskám značně méně. Pro všechno ostatní jenom jednoduše vrátíme `False`:

```
isSmallVowel 'a' = True
isSmallVowel 'e' = True
isSmallVowel 'i' = True
isSmallVowel 'o' = True
isSmallVowel 'u' = True
isSmallVowel _   = False
```

Řeš. 1.3.3 a)

```
logicalAnd :: Bool -> Bool -> Bool
logicalAnd x y = if x then y else False
```

b)

```
logicalAnd' :: Bool -> Bool -> Bool
logicalAnd' True True = True
logicalAnd' _    _    = False
```

Řeš. 1.3.4 Spojnice bodů bude rovnoběžná s osou, jestliže buď $x$ová souřadnice obou bodů bude stejná (pak bude spojnice rovnoběžná s osou x), nebo $y$ová souřadnice bude stejná.

```
parallelToAxis :: (Integer, Integer) -> (Integer, Integer) -> Bool
parallelToAxis (x1, y1) (x2, y2) = x1 == x2 || y1 == y2
```

Řeš. 1.3.5 Řešení pomocí vzorů bývá obvykle více čitelné, zde se zbavíme zanořených podmínek.

```
tell' :: Integer -> String
tell' 1 = "one"
tell' 2 = "two"
tell' n = if mod n 2 == 0 then "(even)" else "(odd)"
```

Řeš. 1.4.1 Typ snadno ověříte pomocí příkazu `:t` k otypování výrazu v GHCi (typ `[Char]` je ekvivalentní typu `String`).

a) `'a' :: Char`; libovolný Unicode znak je stejného typu: `'Ľ'`, `'ř'` nebo speciální znak nového řádku `'\n'`.

b) `"Don't Panic." :: String` (nebo ekvivalentně `[Char]`); `""`, `"a"` nebo `"nejvnějšnější"`.

c) `not :: Bool -> Bool`; např. funkce

```
boolId :: Bool -> Bool
boolId x = x
```

d) `(&&) :: Bool -> Bool -> Bool`; např. `(||)`

e) `True :: Bool`; např. `42 == 6 * 7`

Řeš. 1.4.2 a) `True`, `False`, `not False`, `3 > 3`, `"A" == "c"`, ...
Obecně libovolný správně utvořený výraz z logických hodnot a logických spojek a mnohé další.

b) `-1`, `0`, `42`, ...
Libovolné celé číslo.

c) `3.14`, `2.0e-21`, `2 ** (-4)`, ale také `1`, `42`, ...
Libovolné desetinné číslo, libovolný výraz vracející desetinné číslo, ale také zápis celého čísla může být interpretován jako typu `Double`, pokud to odpovídá kontextu, v němž je vyhodnocen. V interpretu si můžete ověřit, že je výraz otypovatelný na typ `Double` pomocí `:t` *`výraz`* `:: Double`.

d) `False` **není typ**! Jedná se o hodnotu typu `Bool`.

e) `(1, 1)`, `(42, 16)`, `(10 - 5, 10 ^ 10000)`, ...
Libovolná dvojice celých čísel. Pokud jako `Int` zvolíte dostatečně velké číslo pak vám může „přetéct“. Toto rozmezí můžete otestovat zadáním

```
> (minBound, maxBound) :: (Int, Int)
```

do svého interpretu. `Integer` je omezen pouze pamětí počítače.

f) `(0, 3.14, True)`, ...
Trojice, složky musí odpovídat typům.

g) `()`
Takzvaná nultice je typem s jedinou hodnotou. Typ `()` někdy také označujeme jako jednotkový typ nebo v angličtině *unit*. Ačkoli význam takového typu nemusí zatím dávat v Haskellu smysl, časem se s ním setkáme. Nultice je jediným základním typem v Haskellu, kde je typ i hodnota zapisována stejným řetězcem znaků v kódu.

h) `((), (), ())`
Jediná možná hodnota je trojice, jejímž každým prvkem je nultice.

Řeš. 1.4.3 a) `Bool`, výraz je hodnotovým konstruktorem tohoto typu.

b) `String` (ekvivalentně `[Char]`), libovolný výraz v dvojitých uvozovkách je v Haskellu typu `String`.

c) `Bool`, při typování musíme nejprve znát typ funkce `not :: Bool -> Bool` a hodnoty `True :: Bool`. Aplikací funkce se signaturou `Bool -> Bool` na jeden parametr typu `Bool` dostaneme výraz typu `Bool`. Typ prvního parametru v signatuře funkce musí souhlasit s typem reálného prvního parametru při aplikaci, což zde platí.

d) `Bool`, jednotlivé podvýrazy: `(||) :: Bool -> Bool -> Bool`, `True :: Bool`, `False :: Bool`. Typy reálných parametrů odpovídají parametrům v signatuře operátoru `(||)`.

e) Nesprávně utvořený výraz. Jednotlivé podvýrazy: `True :: Bool`, `"" :: String`, `(&&) :: Bool -> Bool -> Bool`. Typ druhého reálného parametru `String` neodpovídá typu druhého parametru signatury, `Bool`. Haskell neprovádí žádné implicitní typové konverze, proto výraz nelze otypovat.

f) `Integer`, výraz `1` může být typu `Integer`, a tedy je možné jej dosadit jako parametr funkce `f`.

g) Nesprávně utvořený výraz. Výraz `3.14` nemůže být typu `Integer`, protože se nejedná o celé číslo, tedy jej nelze dosadit do funkce `f`.

h) `Int`, protože funkce bere dva parametry typu `Int` a `Int` vrací. Výrazy `3` i `8` mohou být `Int`, a tedy je lze dosadit jako parametry.

Řeš. 1.4.4 a) Výraz `not a || b` se uzávorkuje `(not a) || b`, přičemž operátor `(||)` má typ `Bool -> Bool -> Bool`, tedy výraz `not a` má být typu `Bool` a i výraz `b` je typu `Bool`. Nyní stačí jenom určit typ výrazu `a`, který známe z toho, že výraz `not` je typu `Bool -> Bool`, tedy výraz `a` je typu `Bool`. Platí tedy `implication :: Bool -> Bool -> Bool`.

b) Funkce `foo` bere dva argumenty a jejím výsledkem je výraz typu `Bool`. Z prvního řádku předpisu víme, že druhý argument je typu `String`, a ze druhého řádku předpisu víme, že první argument je typu `Char`. Funkce `foo` má tedy typ `Char -> String -> Bool`.

c) Z prvního řádku definice vidíme, že první argument musí být typu `Bool` a funkce taktéž vrací `Bool`. Z druhého řádku pak vidíme, že typ druhého argumentu musí být stejný jako typ návratové hodnoty. Celkově tedy dostáváme `ft :: Bool -> Bool -> Bool`.

Cvičení 2: Rekurze, seznamy, anonymní funkce

Řeš. 2.1.1 Myšlenka řešení může být následující: 0 je sudá, 1 není sudá, a každé jiné kladné číslo je sudé právě tehdy, když číslo o 2 menší je sudé.

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

Řeš. 2.1.2 Myšlenka řešení může být následující: zbytek 0 po dělení třemi je 0, zbytek 1 po dělení třemi je 1, zbytek 2 po dělení třemi je 2 a zbytek dělení třemi pro každé jiné kladné číslo je stejný, jako zbytek po dělení třemi pro číslo o 3 menší.

```
mod3 :: Integer -> Integer
mod3 0 = 0
```

```
mod3 1 = 1
mod3 2 = 2
mod3 x = mod3 (x - 3)
```

Řeš. 2.1.3 Myšlenka je podobná jako v předešlém příkladu. Jen se zde počítá, kolikrát je potřeba odečíst 3, aby se argument dostal do intervalu $\langle 0, 3\rangle$.

```
div3 :: Integer -> Integer
div3 0 = 0
div3 1 = 0
div3 2 = 0
div3 x = 1 + div3 (x - 3)
```

Řeš. 2.1.4

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

Funkce je opět definována po částech. Předpokládáme, že dostane jako argument jenom nezáporné celé číslo. Pokud je argument 0, výsledek je zřejmě opět 0. Pokud je naopak argument kladné číslo, víme, že $n! = n \times (n-1)!$, kde druhý výsledek získáme pomocí rekurzivního volání. Poznamenejme, že závorky kolem `n - 1` je nutno použít, protože jinak by se výraz implicitně uzávorkoval jako `(fact n) - 1`, protože aplikace prefixově zapsané funkce má vyšší prioritu než infixové operátory.

Řeš. 2.1.5 Bázový případ $z^0 = 1$. Pro každé kladné n naopak platí $z^n = z \cdot z^{n-1}$, přičemž hodnota z^{n-1} jde vypočítat rekurzivně.

```
power :: Integer -> Integer -> Integer
power _ 0 = 1
power z n = z * power z (n - 1)
```

Všimněte si, že definovaná funkce vždy použije n rekurzivních volání. Tento počet se ale dá zmenšit, když si uvědomíme, že při každém rekurzivním volání není potřeba n snižovat jen o 1, ale je možné ho zmenšit přibližně na polovinu. Platí totiž

- $z^0 = 1$
- $z^{2n} = (z^n)^2$
- $z^{2n+1} = (z^n)^2 \cdot z$

Následující implementace, která tuto myšlenku využívá, tedy potřebuje jen přibližně $\log(n)$ rekurzivních volání.

```
power' _ 0 = 1
power' z n = if even n then half * half else half * half * z
    where half = power' z (n `div` 2)
```

Řeš. 2.1.6 Stačí si uvědomit, že číslo je mocninou 2 právě tehdy, když je 1 nebo je sudé a jeho polovina je mocninou 2.

```
isPower2 :: Integer -> Bool
isPower2 0 = False
isPower2 1 = True
isPower2 x = even x && isPower2 (div x 2)
```

Řeš. 2.1.7
```
digits :: Integer -> Integer
digits 0 = 0
digits x = x `mod` 10 + digits (x `div` 10)
```

Řeš. 2.1.8 K řešení můžeme použít známý Euklidův algoritmus. Konkrétně použijeme jeho rekurzivní verzi, která využívá zbytky po dělení.

```
mygcd :: Integer -> Integer -> Integer
mygcd x 0 = x
mygcd x y = mygcd (min x y) ((max x y) `mod` (min x y))
```

Řeš. 2.1.9 Příklad lze vyřešit i bez generování všech prvočísel. Stačí zadané číslo dělit 2 tolikrát, kolikrát je to možné, a zároveň přičítat za každé vydělení k výsledku jedničku. Poté číslo budeme dělit 3, 4, 5, 6, atd., dokud po dělení nezbyde výsledek 1. Ačkoliv například čísla 4 a 6 nejsou prvočísla, ale přesto jimi dělíme, není to problém, protože pokud jsme číslo už vydělili 2 a 3, kolikrát to bylo možné, nemůže už být dělitelné ani 4 ani 6.

```
primeDivisors :: Integer -> Integer
primeDivisors n = divisorsFrom n 2
    where
        divisorsFrom 1 _ = 0
        divisorsFrom n d = if mod n d == 0
                           then 1 + divisorsFrom (n `div` d) d
                           else divisorsFrom n (d + 1)
```

Řeš. 2.1.10
```
plus :: Integer -> Integer -> Integer
plus 0 y = y
plus x y = plus (pred x) (succ y)

times :: Integer -> Integer -> Integer
times 0 y = 0
times x 0 = 0
times 1 y = y
times x y = plus y (times (pred x) y)

plus' :: Integer -> Integer -> Integer
plus' x y = if x >= 0
    then plus x y
    else negate (plus (negate x) (negate y))

times' :: Integer -> Integer -> Integer
times' x y = if (x < 0 && y < 0) || (x >= 0 && y >= 0)
    then times (abs x) (abs y)
    else negate (times (abs x) (abs y))
```

Řeš. 2.1.11 Pro zadané číslo n funkce přičte $2n - 1$ k výsledku rekurzivního volání pro vstup o jedna menší. Ten vrátí $2(n-1) - 1$ a součet rekurzivního volání o 1 menší. To se bude dít tak dlouho, než vstup bude 0, pro nějž funkce vrátí 0. Tedy výsledkem bude součet

$$(2n-1) + (2(n-1)-1) + (2(n-2)-1) + \ldots + (2-1) + 0.$$

To lze zapsat přesněji též jako

$$\sum_{i=1}^{n}(2n-1).$$

Předchozí výraz je perfektně správné řešení úlohy. Nicméně s použitím trochy matematiky lze řešení zapsat i elegantněji, aby bylo opravdu vidět, co zadaná funkce počítá.

Odečtení n jedniček lze vytknout za celý součet, a tedy výsledek lze zapsat i jako

$$\left(\sum_{i=1}^{n}2n\right)-n.$$

Násobení dvojkou lze též vytknout před součet, a tedy výsledek lze zapsat i jako

$$2\left(\sum_{i=1}^{n}n\right)-n.$$

Ze střední školy možná víte, že součet aritmetické řady $\sum_{i=1}^{n} n$ je $\frac{n\cdot(n+1)}{2}$. Takže výsledek lze zapsat též jako

$$2\frac{n\cdot(n+1)}{2}-n=n\cdot(n+1)-n=n\cdot n=n^2.$$

Pokud si chcete procvičit látku z Matematických základů informatiky, můžete si zkusit právě odvozenou rovnost

$$\sum_{i=1}^{n}(2n-1)=n^2$$

dokázat matematickou indukcí.

Řeš. 2.2.1 Použijte příkaz `:t` k otypování výrazu v `ghci` (typ `[Char]` je ekvivalentní typu `String`).

a) `[[Char]]` (což je stejné jako `[String]`)
b) `[Char]` (což je stejné jako `String`)
c) `[Char]` (což je stejné jako `String`)
d) `[(Bool, ())]`
e) `[String]`, při otypování takovýchto výrazů je třeba si dát pozor. Výraz sice obsahuje funkci `++`, která má v tomto kontextu typ `String -> String -> String`, avšak `String -> String -> String` není výsledný typ, protože funkci už byly dodány argumenty, a tedy typ prvků v seznamu je `String`.
f) `[Bool -> Bool -> Bool]`
g) `[a]`, z výrazu nevyplývá žádné omezení na typ prvků, který může obsahovat, proto je typ prvků úplně obecný, tedy `a`.
h) `[[a]]`, podobně jako v předešlém případě, žádné omezení na typ prvků vnitřního seznamu.
i) `[[Bool]]` typové omezení vzniká kvůli konkrétní hodnotě ve druhém prvku (prázdný řetězec).

Řeš. 2.2.2 Stačí použít funkci `(:)`.

```
add42 :: [Integer] -> [Integer]
add42 xs = 42 : xs
```

Řeš. 2.2.3 Stačí použít definici funkce podle vzorů. Prázdný seznam je prázdný, žádný jiný seznam není prázdný (duh).

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Řeš. 2.2.4 Opět stačí použít definici funkce podle vzorů. Případy, kdy vstupem funkce je prázdný seznam, buď není potřeba vůbec definovat, nebo lze použít funkci `error` nebo hodnotu `undefined`.

```
myHead :: [a] -> a
myHead (x : _) = x
myHead []      = error "myHead: Empty list."

myTail :: [a] -> [a]
myTail (_ : xs) = xs
myTail []       = error "myTail: Empty list."
```

Řeš. 2.2.5 Stačí si vzpomenout na vzor `x : y : xs` pro seznam, který má alespoň dva prvky.

```
second :: [a] -> a
second (_ : y : _) = y
second _           = error "Your list is too short :("
```

Řeš. 2.2.6

- `[]`
  Tento vzor představuje prázdný seznam. Nemůže reprezentovat žádný z uvedených seznamů.
- `x`
  Na tento vzor se může navázat libovolná hodnota, a tedy zejména libovolný ze zadaných seznamů.
- `[x]`
  Představuje libovolný jednoprvkový seznam. Z uvedených může reprezentovat seznamy `[1]`, `[[]]`, `[[1]]`.
- `[x, y]`
  Představuje libovolný dvouprvkový seznam. Z uvedených může reprezentovat seznamy `[1, 2]`, `[[1], [2, 3]]`.
- `(x : s)`
  Libovolný neprázdný seznam. Proměnná `x` reprezentuje první prvek, proměnná `s` seznam ostatních prvků. Tento vzor může reprezentovat všechny uvedené seznamy (ano, i `[[]]`).
- `(x : y : s)`
  Představuje libovolný seznam, který má alespoň 2 prvky. Proměnná `x` reprezentuje první prvek, `y` druhý prvek a `s` seznam ostatních prvků. Z uvedených může reprezentovat seznamy `[1, 2]`, `[1, 2, 3]`, `[[1], [2, 3]]`.
- `[x : s]`
  Jednoprvkový seznam, jehož jediným prvkem je neprázdný seznam. Proměnná `x` reprezentuje první prvek vnitřního seznamu, proměnná `s` seznam ostatních prvků vnitřního seznamu. Z uvedených může reprezentovat pouze seznam `[[1]]`.
- `((x : y) : s)`
  Představuje neprázdný seznam, jehož prvním prvkem je neprázdný seznam. Proměnné `x` a `y` reprezentují první prvek prvního prvku a seznam ostatních prvků prvního prvku, proměnná `s` reprezentuje ostatní prvky vnějšího seznamu. Z uvedených může reprezentovat seznamy `[[1]]`, `[[1], [2, 3]]`.

Řeš. 2.2.7 Případ prázdného seznamu nemusíme řešit. Pro jednoprvkový seznam vrátíme rovnou jeho poslední prvek:

```
getLast [x] = x
```

Všechny zbývající případy seznamů mají alespoň dva prvky. Jednoduchá úvaha vede k tomu, že poslední prvek seznamu, který má alespoň dva prvky, je stejný jako poslední prvek téhož seznamu, ale bez prvního prvku. Tedy ze vstupního seznamu odstraníme první prvek a na zbytek aplikujeme rekurzivně funkci `getLast`:

```
getLast (x : xs) = getLast xs
```

Řeš. 2.2.8 Funkci definujeme obdobně jako funkci `getLast`. Začneme jednoprvkovým seznamem, kdy výsledkem je prázdný seznam:

```
stripLast [x] = []
```

Všechny zbývající případy seznamů mají dva nebo více prvků. V takovém případě bude první prvek zadaného seznamu určitě ve výsledném seznamu a zbytek lze vypočítat rekurzivně:

```
stripLast (x : xs) = x : stripLast xs
```

Srovnejte s definicí funkce `getLast`.

Řeš. 2.2.9 Délka prázdného seznamu je 0. Délka alespoň jednoprvkového seznamu je o 1 větší, než délka vstupního seznamu bez prvního prvku.

```
len :: [a] -> Integer
len []       = 0
len (_ : xs) = 1 + len xs
```

Výpočet této funkce probíhá například takto:

```
len (1 : (2 : [])) ⇝ 1 + len (2 : []) ⇝ 1 + (1 + len [])
    ⇝ 1 + (1 + 0) ⇝* 2
```

Řeš. 2.2.10 Prázdný seznam neobsahuje žádné číslo. Neprázdný seznam obsahuje zadané číslo právě tehdy, když je ono číslo prvním prvkem zadaného seznamu, nebo ho obsahuje zbytek zadaného seznamu.

```
containsNumber :: Integer -> [Integer] -> Bool
containsNumber _ [] = False
containsNumber e (x : xs) = e == x || containsNumber e xs
```

Řeš. 2.2.11 Myšlenka je podobná jako u funkce `containsNumber`, jen je potřeba si počítat, kolikrát zadané číslo ještě chceme vidět. Pokud se dostaneme na 0, číslo už jsme viděli dostatečněkrát, a vrátíme tedy `True`.

```
containsNNumbers :: Integer -> Integer -> [Integer] -> Bool
containsNNumbers 0 _ _        = True
containsNNumbers _ _ []       = False
containsNNumbers n x (y : ys) = if x == y
    then containsNNumbers (n - 1) x ys
    else containsNNumbers n x ys
```

Řeš. 2.2.12 Funkce `getPoints` je podobná funkci `containsNumber`, ale místo logické hodnoty budeme vracet příslušnou hodnotu.

```
getPoints :: String -> [(String, Integer)] -> Integer
getPoints _ [] = 0
```

```
getPoints wanted ((name, points) : xs) = if wanted == name
    then points
    else getPoints wanted xs
```

U funkce `getBest` se hodí definovat si pomocnou funkci, která jako další argument dostane i jméno a počet bodů aktuálně nejlepšího studenta. Tohoto aktuálně nejlepšího studenta bude v průběhu vypočtu měnit a na konci seznamu ho vrátí.

```
getBest :: [(String, Integer)] -> String
getBest (x : xs) = fst (getBestWithDefault x xs)
    where
        getBestWithDefault current [] = current
        getBestWithDefault (curN, curP) ((newN, newP) : xs) =
            if newP > curP
            then getBestWithDefault (newN, newP) xs
            else getBestWithDefault (curN, curP) xs
```

Řeš. 2.2.13

```
nth :: Integer -> [a] -> a
nth 0 (x : _)  = x
nth n (_ : xs) = nth (n - 1) xs
```

Řeš. 2.2.14 Nejjednodušší je funkci definovat podle vzoru na prvním argumentu. Pokud je první seznam prázdný, výsledkem je přímo druhý seznam. Pokud je první seznam neprázdný, výsledný seznam obsahuje první prvek prvního seznamu a pak zřetězení zbytku prvního seznamu s druhým seznamem.

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x : xs) ys = x : append xs ys
```

Řeš. 2.2.15 Zde se hodí vzor pro seznamy délky alespoň dva, protože potřebujeme pojmenovat první dva prvky vstupního seznamu. Poté stačí udělat z nich dvojici a zbytek seznamu vyřešit rekurzivně.

```
pairs :: [a] -> [(a, a)]
pairs (x : y : s) = (x, y) : pairs s
pairs  _          = []
```

Řeš. 2.2.16

a) Součet prázdného seznamu je 0. Součet neprázdného seznamu je součet prvního prvku a součtu zbytku seznamu.

```
listSum :: [Integer] -> Integer
listSum []       = 0
listSum (x : xs) = x + listSum xs
```

b) Existují nejméně dva přístupy k řešení, pokud nechceme explicitně pracovat s délkou seznamu. Jeden z nich je, že využijeme vzoru `(x : y : zs)`, který bere ze seznamu *po dvou* prvcích, tím pádem víme, že pokud tak skončíme na jednom prvku, seznam musel obsahovat lichý počet prvků:

```
oddLength :: [Integer] -> Bool
oddLength [] = False
oddLength [_] = True
oddLength (_ : _ : zs) = oddLength zs
```

Druhý přístup k řešení je, že odpověď postupně *vyskládáme* z prázdného seznamu, protože víme, že ten obsahuje sudý počet prvků. Každým dalším prvkem odpověď

změníme na opačnou, s posledním prvkem získáme odpověď pro celý seznam:

```
oddLength' :: [Integer] -> Bool
oddLength' [] = False
oddLength' (_ : xs) = not (oddLength xs)
```

c) Opět přímočará rekurzivní definice funkce podle vzorů.

```
add1 :: [Integer] -> [Integer]
add1 [] = []
add1 (x : xs) = (x + 1) : add1 xs
```

d) Opět přímočará rekurzivní definice funkce podle vzorů.

```
multiplyN :: Integer -> [Integer] -> [Integer]
multiplyN _ [] = []
multiplyN n (x : xs) = (n * x) : multiplyN n xs
```

e) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteEven :: [Integer] -> [Integer]
deleteEven [] = []
deleteEven (x : xs) = if even x
    then deleteEven xs
    else x : deleteEven xs
```

f) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteElem :: Integer -> [Integer] -> [Integer]
deleteElem _ [] = []
deleteElem n (x : xs) = if x == n
    then deleteElem n xs
    else x : deleteElem n xs
```

g) Opět přímočará rekurzivní definice funkce podle vzorů.

```
largestNumber :: [Integer] -> Integer
largestNumber [x] = x
largestNumber (x : xs) = x `max` largestNumber xs
```

h) Stačí si uvědomit, jaké všechny případy mohou nastat. Jediný zajímavý případ je, když oba vstupní seznamy jsou neprázdné. V takovém případě je potřeba porovnat první prvky obou seznamů a také rekurzivně porovnat zbytky obou seznamů.

```
listsEqual :: [Integer] -> [Integer] -> Bool
listsEqual [] [] = True
listsEqual [] _ = False
listsEqual _ [] = False
listsEqual (x : xs) (y : ys) = x == y && listsEqual xs ys
```

i) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
multiplyEven :: [Integer] -> [Integer]
multiplyEven [] = []
multiplyEven (x : xs) = if even x
    then (2 * x) : multiplyEven xs
    else multiplyEven xs
```

j) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
sqroots :: [Double] -> [Double]
sqroots [] = []
sqroots (x : xs) = if x > 0
```

```
        then sqrt x : sqroots xs
        else sqroots xs
```

Řeš. 2.2.17

```
everyNth :: Integer -> [a] -> [a]
everyNth n xs = everyNthOffset n xs 0
    where
        everyNthOffset _ [] _ = []
        everyNthOffset n (x : xs) 0 = x : everyNthOffset n xs (n - 1)
        everyNthOffset n (x : xs) m = everyNthOffset n xs (m - 1)
```

Řeš. 2.2.18

```
brackets :: String -> Bool
brackets s = bracketsWithDiff s 0
    where
        bracketsWithDiff [] k = k == 0
        bracketsWithDiff ('(' : xs) k = bracketsWithDiff xs (k + 1)
        bracketsWithDiff (')' : xs) k = k > 0 &&
                                        bracketsWithDiff xs (k - 1)
```

Řeš. 2.2.19 Funkci, která rozhodne, jestli je řetězec palindromem, zadefinujeme jednoduše pomocí funkce `reverse` a porovnání.

```
palindrome :: String -> Bool
palindrome str = str == reverse str
```

Po krátkém zamyšlení zjistíme, že na doplnění slova na palindrom nám stačí najít nejdelší příponu slova, která tvoří palindrom. Vynechané znaky ze začátku pak doplníme i na konec řetězce v obráceném pořadí.

```
palindromize :: String -> String
palindromize s = if palindrome s
    then s
    else [head s] ++ palindromize (tail s) ++ [head s]
```

Poznámka: Vzhledem k častému využívání sekvenčního spojování seznamů (`++`) nemá tato funkce optimální časovou složitost. Zkuste se zamyslet, jak by se dala napsat efektivnější funkce.

Řeš. 2.2.20

```
getMiddle :: [a] -> a
getMiddle xs = tortoiseRabbit xs xs
    where
        tortoiseRabbit (t : _) [_] = t
        tortoiseRabbit (t : _) [_, _] = t
        tortoiseRabbit (_ : ts) (_ : _ : rs) = tortoiseRabbit ts rs
```

Řeš. 2.3.2

```
getNames :: [(String, Integer)] -> [String]
getNames s = map fst s

successfulRecords :: [(String, Integer)] -> [(String, Integer)]
successfulRecords s = filter successful s
    where
        successful (_, p) = p >= 50

successfulNames :: [(String, Integer)] -> [String]
successfulNames s = getNames (successfulRecords s)
```

```
successfulStrings :: [(String, Integer)] -> [String]
successfulStrings s = map formatStudent (successfulRecords s)
    where
        formatStudent (n, p) = n ++ ": " ++ show p ++ " b"
```

Řeš. 2.3.3 Funkci `map` lze využít v případě, že potřebujeme jistým způsobem modifikovat každý prvek zadaného seznamu.

```
add1' :: [Integer] -> [Integer]
add1' xs = map plus1 xs
    where plus1 x = x + 1

multiplyN' :: Integer -> [Integer] -> [Integer]
multiplyN' n xs = map timesN xs
    where timesN x = x * n
```

Funkci `filter` naopak použijeme, chceme-li ze vstupního seznamu vybrat pouze některé prvky.

```
deleteEven' :: [Integer] -> [Integer]
-- chci odstranit sudá čísla, ponechám tedy ty prvky,
-- pro které platí odd (číslo je liché)
deleteEven' xs = filter odd xs

deleteElem' :: Integer -> [Integer] -> [Integer]
deleteElem' n xs = filter notEqualN xs
    where notEqualN x = x /= n
```

Funkce `map` a `filter` lze vhodně kombinovat, pokud chci prvky modifikovat a zároveň filtrovat.

```
multiplyEven' :: [Integer] -> [Integer]
multiplyEven' xs = map times2 (filter even xs)
    where times2 x = x * 2

sqroots' :: [Double] -> [Double]
sqroots' xs = map sqrt (filter greaterThan0 xs)
    where greaterThan0 x = x > 0
```

Zbývající funkce nelze vhodně implementovat pomocí `map` a `filter`: `listSum`, `oddLength` a `listsEqual` se vyhodnocují na jeden prvek (typu `Integer` nebo `Bool`), ale `map` a `filter` vrací seznamy. Funkce `listsEqual` musí najednou procházet dva seznamy, ale `map` a `filter` rekurzivně prochází vždy pouze jeden seznam (lze elegantně řešit použitím funkce `zipWith`, je však potřeba ohlídat, zda mají seznamy stejnou délku).

Řeš. 2.3.4

```
import Data.Char
toUpperStr :: String -> String
toUpperStr = map toUpper
```

Řeš. 2.3.5 Nejdřív si zadefinujeme pomocný predikát `isvowel`, který o znaku určí, jestli je samohláskou. Následně jednotlivé řetězce projdeme funkcí `filter`.

```
isvowel :: Char -> Bool
isvowel c = elem (toUpper c) "AEIOUY"
vowels :: [String] -> [String]
```

```
vowels s = map (filter isvowel) s
```

Řeš. 2.3.7
```
assignPrizes :: [String] -> [Integer] -> [(String, Integer)]
assignPrizes = zip

formatPrizeText :: String -> Integer -> String
formatPrizeText n p = n ++ ": " ++ show p ++ " Kc"

prizeTexts :: [String] -> [Integer] -> [String]
prizeTexts ns ps = zipWith formatPrizeText ns ps
```

Řeš. 2.3.8
```
neighbors :: [a] -> [(a, a)]
neighbors xs = zip xs (tail xs)
```

Řeš. 2.3.9
```
f1 :: [Integer] -> Bool
f1 (x : y : s) = x == y || f1 (y : s)
f1 _           = False
```

Nebo kratší řešení používající funkci `zipWith` a funkci `or`, která spočítá logický součet všech hodnot v zadaném seznamu:

```
f2 :: [Integer] -> Bool
f2 s = or (zipWith (==) s (tail s))
```

Řeš. 2.3.10
```
myMap :: (a -> b) -> [a] -> [b]
myMap f [] = []
myMap f (x : xs) = f x : myMap f xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFilter p (x : xs) = if p x
    then x : myFilter p xs
    else myFilter p xs

myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith f (x : xs) (y : ys) = f x y : myZipWith f xs ys
myZipWith _ _ _ = []
```

Řeš. 2.3.11 Jedná se o funkce `any` a `all`.
http://hackage.haskell.org/package/base/docs/Prelude.html#v:any
http://hackage.haskell.org/package/base/docs/Prelude.html#v:all

Řeš. 2.3.12 http://hackage.haskell.org/package/base/docs/Prelude.html#v:takeWhile

Řeš. 2.4.4
```
quickSort :: [Integer] -> [Integer]
quickSort [] = []
quickSort [x] = [x]
quickSort (x : xs) =
    quickSort (filter (\y -> y < x) xs) ++
    [x] ++
    quickSort (filter (\y -> y >= x) xs)
```

Cvičení 3: Manipulace s funkcemi, typy

Řeš. 3.1.1

a) `Bool`
b) `Bool -> Bool -> Bool`
c) `[a]`
d) `String` neboli `[Char]`
e) `[(Bool, String)]`
f) Nelze otypovat, seznamy musí být homogenní (všechny hodnoty musí mít stejný typ).

Řeš. 3.1.2

a) `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`.
b) `Num a => a`
c) Nelze otypovat. Chybová hláška `No instance for (Num Bool) arising from the literal '1'` říká, že `Bool` není číslo (instance `Num`).
d) `(Ord a, Num a) => [a]` (obě části kontextu jsou nutné, mezi `Ord` a `Num` není žádný vztah implikace)
e) `(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`.
f) `(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.
g) `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. 3.1.3

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)` $\sim$ `(c, d)` a `(b, a)` $\sim$ `(c, d)`), jediná možnost je, že oba typy ve dvojici budou stejné. `maybeSwap :: 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. 3.1.4 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`, tu 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. 3.1.5 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 :: Ord a => [a] -> (a, a)`

Nejprve si musíme určit typ pomocné funkce `mm'`. Z prvního řádku (který je zároveň

bází rekurze) vidíme vztah mezi jejími návratovými hodnotami a prvními dvěma argumenty a také, že třetí argument musí být seznam: `a -> b -> [c] -> (a, b)`. Z druhého řádku a z typu funkcí `min` a `max` pak vidíme, že první dva argumenty mají stejný typ, který je zároveň stejný jako typ položek v seznamu v třetím argumentu, dostáváme tedy `mm' :: Ord a => a -> a -> [a] -> (a, a)` (kontext plyne z použití `min` a `max`).
V samotné definici funkce `mm` pak není první řádek příliš zajímavý, jeho typ je `[a] -> b` (protože `error :: String -> b`). Pro typ druhého řádku už jen stačí dosadit za první dva argumenty v typu `mm'`.
Funkce `mm` počítá minimum a maximum z hodnot v seznamu a dělá to v jednom průchodu.

Řeš. 3.1.6 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. 3.2.1 a) `(+) 2 40` $\rightsquigarrow^*$ `42`
Funkce, která k dané hodnotě přičte zleva číslo 2.

b) `(* 2) 21` $\rightsquigarrow^*$ `42`
Funkce, která dané číslo vynásobí zprava dvěma.

c) `(- 2)` $\rightsquigarrow$ `-2`
Číslo -2. Jelikož `-` je binární i unární operátor, nelze jej použít v pravé operátorové sekci. Místo toho však existuje funkce `subtract`: `subtract 2 44` $\rightsquigarrow^*$ `42`.

d) `(2 -) 1` $\rightsquigarrow^*$ `1`
Funkce, která dané číslo odečte od dvojky.

e) Syntakticky špatně utvořený výraz. Operátorové sekce musí být vždy v závorkách.

Řeš. 3.2.2

a)
```
upper' :: String -> String
upper' xs = map toUpper xs
```

b)
```
embrace' :: [String] -> [String]
embrace' xs = map ('[' :) (map (++ "]") xs)
```

Nebo lze pro `(:)` použít prefixový tvar:

```
embrace'' :: [String] -> [String]
embrace'' xs = map ((:) '[') (map (++ "]") xs)
```

c)
```
sql' :: (Ord a, Num a) => [a] -> a -> [a]
sql' xs lt = map (^ 2) (filter (< lt) xs)
```

Řeš. 3.2.3

a) První výraz je díky implicitním závorkám částečné aplikace ekvivalentní `((f 1) g) 2` a odpovídá funkci `f` beroucí tři parametry a druhý je ekvivalentní `(f 1) (g 2)`.

b) Ano, `(f 1 g) 2` ≡ `f 1 g 2` ≡ `(f 1) g 2` (tedy funkce `f` tu bere dva argumenty).

c) Ne, `(* 2) 3` ≡ `(*) 3 2` ≡ `3 * 2`. Neexistuje pravidlo, které by zaručovalo, že `3 * 2` se bude rovnat `2 * 3` (standard jazyka Haskell komutativitu operátoru `(*)` nevynucuje). Nezapomínejme, že všechny operátory definované typovými třídami můžeme předefinovat. *Poznámka*: (pokročilejší) Toto by bylo možné pouze v případě, že by komutativity vyžadovaly axiomy typové třídy, ve které je daný operátor/funkce definována. Ani to by však nezaručovalo skutečnou korektnost – interpret/kompilátor platnost axiomů nekontroluje (ani to není v jeho silách). Zůstává pouze důvěra v programátora, že jeho implementace je korektní.

d) Ano, `(* 3)` je pravá sekce.

Řeš. 3.3.1

a)
```
map even :: Integral a => [a] -> [Bool]
```

b)
```
map head . snd :: (a, [[b]]) -> [b]
```

c)
```
filter ((4 >) . maximum) :: (Ord a, Num a) => [[a]] -> [[a]]
```

d)
```
const const :: a -> b -> c -> b
```

Řeš. 3.3.2

a)
```
failing' :: [(Int, Char)] -> [Int]
failing' sts = map fst (filter ((== 'F') . snd) sts)

failing'' :: [(Int, Char)] -> [Int]
failing'' = map fst . (filter ((== 'F') . snd))
```

b)
```
embraceWith' :: Char -> Char -> [String] -> [String]
embraceWith' l r = map ((l :) . (++ [r]))
```

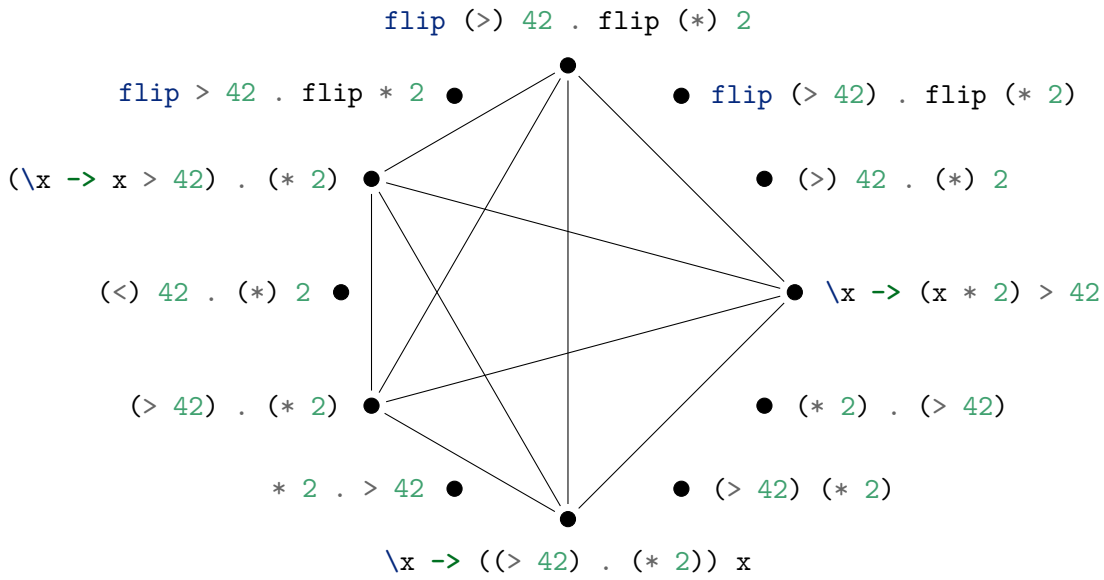
Argumenty `l` a `r` nelze rozumně odstranit.

c)
```
divisibleBy7' :: [Integer] -> [Integer]
divisibleBy7' = filter ((== 0) . (`mod` 7))
```

d)
```
letterCaesar' :: String -> String
letterCaesar' = map (chr . (3 +) . ord) . filter isLetter
```

e)
```
zp' :: (Integral a, Num b) => [a] -> [b] -> [b]
zp' = zipWith (flip (^))
```

Řeš. 3.3.3 Vzájemně ekvivalentní funkce jsou spojeny:



Všechny vzájemně ekvivalentní výrazy můžeme získat různými ekvivalentními úpravami z `\x -> (x * 2) > 42`.

Zbývající výrazy:

- `(* 2) . (> 42)` by nejprve porovnával vstup s hodnotou `42` a poté teprve přičítal `2` k výsledku typu `Bool`, je tedy typově nesprávný.
- `(> 42) (* 2)` aplikuje sekci `(> 42)` na `(* 2)`, `(> 42)` však vyžaduje na vstupu číslo, ale `(* 2)` je typu `Num a => a -> a`. Výraz je tedy typově nesprávný.
- `* 2 . > 42` je syntakticky nesprávný, operátorové sekce je vždy potřeba uzávorkovat.
- `(<) 42 . (*) 2` není nutně ekvivalentní pro všechny vstupy, protože nic negarantuje, že `(*)` je komutativní a při prohození argumentů lze zaměnit `(>)` za `(<)`. Jelikož jsou tyto funkce definovány zvlášť pro každý datový typ v dané typové třídě, je možné, že nějaká implementace toto splňovat nebude.
- `flip > 42 . flip * 2` se uzávorkuje jako `flip > ((42 . flip) * 2)`, a pokouší se tedy skládat číslo 42 a funkci `flip` a dokonce tento výsledek složení násobit dvěma. Tento výraz je tedy typově nesprávný.
- `flip (> 42) . flip (* 2)` – `flip` očekává binární funkci, ale `(> 42)` a `(* 2)` jsou nutně unární.
- `(>) 42 . (*) 2` je ekvivalentní s `\x -> 42 > (2 * x)`, argumenty jsou tedy otočeny.

Řeš. 3.3.4 a) Naším cílem je ze zadané funkce vytvořit negovanou funkci. Z typu funkce `negp` vidíme, že můžeme uvést dva argumenty – predikát a hodnotu. Pak jen na výsledek volání `f` zavoláme funkci `not`, která realizuje logickou negaci.

```
negp :: (a -> Bool) -> a -> Bool
negp f x = not (f x)
```

b) Funkci z předchozího příkladu můžeme přepsat do tvaru složení funkcí:

```
negp f x = (not . f) x
```

Odtud můžeme následně odstranit formální argument:

```
negp f = not . f
```

K tomuto výsledku můžeme dojít i přímo, uvědomíme-li si, že negace predikátu je složením predikátu s funkcí negace.

c) Dále lze tělo funkce přepsat do prefixového tvaru:

```
negp f = (.) not f
```

A následně lze odstranit poslední formální argument `f`, čímž dostaneme definici plně bez formálních argumentů:

```
negp = (.) not
```

Alternativně lze tělo funkce upravit pomocí operátorové sekce:

```
negp f = (not .) f
negp = (not .)
```

Poznámka: Z hlediska elegance a čistoty kódu by byla většinou programátorů v Haskellu pravděpodobně preferována varianta `negp f = not . f`.

Řeš. 3.3.6 a)
```
\x -> (f . g) x
f . g
```

b)
```
\x -> f . g x
\x -> (.) f (g x)
\x -> ((.) f . g) x
(.) f . g
```

c)
```
\x -> f x . g
\x -> (.) (f x) g
\x -> flip (.) g (f x)
\x -> (flip (.) g . f) x
flip (.) g . f
```

Řeš. 3.3.7 a)
```
(^ 2) . mod 4 . (+ 1)
\x -> ((^ 2) . mod 4 . (+ 1)) x
\x -> (^ 2) (mod 4 ((+ 1) x))
\x -> (mod 4 (x + 1)) ^ 2
```

b)
```
(+) . sum . take 10
\x -> ((+) . sum . take 10) x
\x -> (+) (sum (take 10 x))
\x y -> (+) (sum (take 10 x)) y
\x y -> sum (take 10 x) + y
```

c)
```
map f . flip zip [1, 2, 3]
\x -> (map f . flip zip [1, 2, 3]) x
\x -> map f (flip zip [1, 2, 3] x)
\x -> map f (zip x [1, 2, 3])
```

d)
```
(.)
\f g -> (.) f g
\f g -> f . g
\f g x -> (f . g) x
\f g x -> f (g x)
```

Řeš. 3.3.8 a)
```
f :: a -> b -> b
f x y = y
f x y = const y x
f x y = flip const x y
f = flip const
```
b)
```
f :: Num a => a -> b -> a
f x y = const (3 + x) y
f x = const (3 + x)
f x = const ((3 +) x)
f x = (const . (3 +)) x
f = const . (3 +)
```

Řeš. 3.3.9 a)
```
\_ -> x
\t -> x
\t -> const x t
const x
```

b)
```
\x -> f x 1
\x -> flip f 1 x
flip f 1
```

c)
```
\x -> f 1 x True
\x -> (f 1) x True
\x -> flip (f 1) True x
flip (f 1) True
```

d)
```
const x
```

e)
```
\x -> 0
\x -> const 0 x
const 0
```

f) Není možno převést, poněvadž `if ... then ... else ...` není klasická funkce, ale syntaktická konstrukce, podobně jako `let ... in ...`.

g)
```
\f -> flip f x
\f -> flip flip x f
flip flip x
```

Řeš. 3.3.10 a) Postupně převádíme:
```
f1 x y z = x
f1 x y z = const x z -- přidáme z tak, abychom ho mohli odstranit
f1 x y = const x
f1 x y = const (const x) y -- přidáme y
f1 x = const (const x)
f1 x = (const . const) x
f1 = const . const
```
b)
```
f2 x y z = y
f2 x y z = const y z
f2 x = const -- eta-redukujeme obojí
f2 x = const const x -- přidáme x
```

```
f2 = const const
```

c)
```
f3 x y z = z
f3 x y z = id z -- přidáme uměle identitu
f3 x y = id
f3 x y = const id y -- přidáme y
f3 x = const id
f3 x = const (const id) x -- přidáme x
f3 = const (const id)
```

Řeš. 3.3.11 Několikrát po sobě použijeme funkci `flip`.
`g = flip (flip ... (flip (flip f c1) c2) ... cn)`

Cvičení 4: Vlastní a rekurzivní datové typy, `Maybe`

Řeš. 4.1.1 a) Příklady hodnot jsou:

```
Cube 1 2 3
Cylinder (-3) (1/2)
```

Některé z těchto hodnot sice nemusí odpovídat skutečným tělesům, ale uvedený datový typ je umožňuje zapsat.

b) Hodnotové konstruktory jsou umístěny jako první identifikátor ve výrazech oddělených svislítky. Tedy v tomto případě to jsou `Cube`, `Cylinder`. Také hodnotový konstruktor začíná velkým písmenem.

c) Typové konstruktory můžeme rozlišit na nově definované a na ty, které jsou jenom použité. Typový konstruktor je vždy umístěn jako první identifikátor za klíčovým slovem `data`, tedy v tomto případě `Object`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Float`.

d) Funkci budeme definovat po částech. Pro každý možný tvar hodnoty typu `Object`, tj. pro každý typ tělesa definujeme funkci osobitě. Poznamenejme, že je nutné použít závorky kolem argumentů funkcí, aby byl tento výraz považován jako jeden argument, ne jako několik argumentů. K definici funkcí můžeme využít i konstantu `pi`, která je v Haskellu standardně dostupná.

```
volume :: Object -> Float
volume (Cube x y z)   = x * y * z
volume (Cylinder r v) = pi * r * r * v

surface :: Object -> Float
surface (Cube x y z)   = 2 * (x * y + x * z + y * z)
surface (Cylinder r v) = 2 * pi * r * (v + r)
```

Řeš. 4.1.2
```
weekend :: Day -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False
```

Pokud je typ `Day` zaveden v typové třídě `Eq`, můžeme použít i následující alternativní definici funkce `weekend`:

```
weekend' :: Day -> Bool
weekend' d = d == Sat || d == Sun
```

Řeš. 4.1.3

```
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. 4.1.4

```
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. 4.1.5

```
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. 4.1.6

```
data Jar = EmptyJar
         | Cucumbers
         | Jam String
         | Compote Int
           deriving (Show, Eq)
today :: Int
today = 2020

stale :: Jar -> Bool
stale EmptyJar    = False
stale Cucumbers   = False
stale (Jam _)     = False
stale (Compote x) = today - x >= 10
```

Alternativní řešení s menším počtem vzorů a hlavně přehlednějším zápisem (víme, že výsledek pro první tři případy je stejný):

```
stale' :: Jar -> Bool
stale' (Compote x) = today - x >= 10
stale' _           = False
```

Řeš. 4.1.7

a) Nulární typový konstruktor `X`, unární hodnotový konstruktor `Value`.
b) Nulární typové konstruktory `M` a `N`, nulární hodnotové konstruktory `A`, `B`, `C`, `D`, unární hodnotové konstruktory `N` a `M`.
c) Chybná deklarace: `Hah` je v seznamu použito jako typový konstruktor, jedná se však o hodnotový konstruktor.
d) Nulární typový konstruktor `FNM`, ternární hodnotový konstruktor `T`.
e) Vytváří se pouze typové synonymum (nulární).
f) Nulární typový konstruktor `E`, unární hodnotový konstruktor `E`

Řeš. 4.1.8

a) Ok.
b) Nok, typová proměnná `a` musí být argumentem konstruktoru `Makro`.
c) Nok, hodnotový konstruktor není možné použít vícekrát.
d) Nok, typová proměnná `c` musí být argumentem konstruktoru `Fun`.
e) Nok, argumenty konstruktoru `Fun` mohou být pouze typové proměnné, ne složitější typové výrazy (tj. není možné použít definici podle vzoru).
f) Nok, `Z X` není korektní výraz, protože `X` je hodnotový konstruktor.
g) Nok, každý hodnotový konstruktor musí začínat velkým písmenem.
h) Nok, výraz je interpretován jako hodnotový konstruktor `Makro` se třemi argumenty: `Int`, `->` a `Int` – je nutné přidat závorky kolem `(Int -> Int)`.
i) Nok, syntax výrazu je chybná: `type` musí mít na pravé straně pouze jednu možnost (jedná se o typové synonymum, ne o nový datový typ).
j) Ok, typový i hodnotové konstruktory mají stejný název. Na pravé straně definice je `X` nejdřív binárním hodnotovým a pak dvakrát nulárním typovým konstruktorem. Jediná plně definovaná hodnota tohoto typu je `x = X x x`.

Řeš. 4.1.9

a) Matematická definice rovnosti zlomků nám říká, že $\frac{a}{b} = \frac{c}{d} \Leftrightarrow a \cdot d = b \cdot c$. Funkci pak už snadno postavíme na této rovnosti.

```
fraceq :: Frac -> Frac -> Bool
fraceq (a, b) (c, d) = a * d == b * c
```

b) Zde opět použijeme vestavěnou funkci `signum`.

```
nonneg :: Frac -> Bool
nonneg (a, b) = signum a == signum b
```

c) K řešení nám opět pomůže si nejdříve matematicky zapsat požadovaný výraz: $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$.

```
fracplus :: Frac -> Frac -> Frac
fracplus (a, b) (c, d) = simplify (a * d + b * c, b * d)
```

d)
```
fracminus :: Frac -> Frac -> Frac
fracminus (a, b) (c, d) = fracplus (a, b) (-c, d)
```

e)
```
fractimes :: Frac -> Frac -> Frac
fractimes (a, b) (c, d) = simplify (a * c, b * d)
```

f)
```
fracdiv :: Frac -> Frac -> Frac
fracdiv (_, _) (0, _) = error "division by zero"
```

```
fracdiv (a, b) (c, d) = fractimes (a, b) (d, c)
```

g) Pro úpravu do základního tvaru postačí vydělit čitatele i jmenovatele jejich největším společným jmenovatelem (můžeme použít vestavěnou funkci `gcd`, která pracuje i se zápornými čísly). Musíme si však dát pozor na znaménko – základním tvarem zlomku $\frac{-2}{-4}$ je $\frac{1}{2}$ a nikoliv $\frac{-1}{-2}$. To můžeme zajistit následovně: číslo ve jmenovateli základního tvaru budeme mít vždy kladné a znaménko přeneseme do čitatele (například pomocí vestavěné funkce `signum`).

```
simplify :: Frac -> Frac
simplify (a, b) = ((signum b) * (a `div` d), abs (b `div` d))
    where d = gcd a b
```

Poznámka: Haskell má vestavěný typ pro zlomky, `Rational`. Ten je reprezentován v podstatě stejným způsobem, tedy jako dvě hodnoty typu `Integer`. Nicméně nejedná se přímo o dvojice, ale typ `Rational` definuje hodnotový konstruktor `%`, tedy například zlomek $\frac{1}{4}$ zapíšeme jako `1 % 4`. Datový typ `Rational` je instancí mnoha typových tříd, mimo jiné `Num` a `Fractional`, proto s ním lze pracovat jako s jinými číselnými typy v Haskellu a používat operátory jako `(+)`, `(-)`, `(*)`, `(/)`.

Řeš. 4.1.10

```
commonPricing :: Int -> Float
commonPricing cups = fromIntegral (13 * pricingCups)
    where
        pricingCups = cups - cups `div` 10

employeeDiscount :: Float -> Float
employeeDiscount basePrice = basePrice * (1 - 0.15)

studentPricing :: Int -> Float
studentPricing cups = fromIntegral cups

data PricingType = Common | Employee | Student

computePrice :: PricingType -> Int -> (Int -> Float)
             -> (Float -> Float) -> (Int -> Float) -> Float
computePrice Common cups cp _ _     = cp cups
computePrice Employee cups cp ed _ = (ed . cp) cups
computePrice Student cups _ _ sp    = sp cups
```

Řeš. 4.1.11

```
commonPricing :: Int -> Float
commonPricing cups = fromIntegral (13 * pricingCups)
    where
        pricingCups = cups - cups `div` 10

employeeDiscount :: Float -> Float
employeeDiscount basePrice = basePrice * (1 - 0.15)

studentPricing :: Int -> Float
studentPricing cups = fromIntegral cups

data PricingType' = Common'
                  | Special (Int -> Float)
                  | Discount (Float -> Float)
```

```
computePrice :: PricingType' -> Int -> (Int -> Float) -> Float
computePrice Common' cups cp        = cp cups
computePrice (Special sp) cups _    = sp cups
computePrice (Discount dp) cups cp = dp (cp cups)

common :: PricingType'
common = Common'

employee :: PricingType'
employee = Discount employeeDiscount

student :: PricingType'
student = Special studentPricing
```

Řeš. 4.2.1

a) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu.
b) Korektní typ s hodnotou například `Nothing`.
c) Nekorektní výraz – hodnotový konstruktor `Just` očekává hodnotu, `a` je ovšem typ.
d) Nekorektní výraz – hodnotový konstruktor `Just` je aplikovaný na příliš mnoho argumentů. Pro úplnost dodejme, že výraz `Just (Just 2)` by byl korektní hodnotou typu `Num a => Maybe (Maybe a)`.
e) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu `Nothing`.
f) Korektní hodnota typu `Maybe (Maybe a)`.
g) Nekorektní výraz – nulární hodnotový konstruktor `Nothing` nebere žádné argumenty.
h) Nekorektní výraz – jedna hodnota je typu `(Num a) => Maybe a`, druhá typu `Maybe (Maybe b)`.
i) Korektní hodnota typu `(Num a) => Maybe [Maybe a]`.
j) Korektní hodnota typu `(Num a) => Maybe (a -> a)`.
k) Korektní hodnota typu `Bool -> Maybe a -> Maybe a`.
l) Korektní hodnota (funkce) typu `a -> Maybe a`.
m) Korektní hodnota (ne funkce) typu `Maybe (a -> Maybe a)`.
n) Nekorektní výraz – implicitní závorky jsou `(Just Just) Just` a podle předchozího příkladu víme, že `Just Just :: Maybe (a -> Maybe a)`. Avšak tento výraz není funkcí (je to `Maybe` výraz – podstatný je vnější typový konstruktor), a proto ho nemůžeme aplikovat na hodnotu, jako by to byla funkce.

Řeš. 4.2.2

```
safeDiv :: Integral a => a -> a -> Maybe a
safeDiv x 0 = Nothing
safeDiv x y = Just (x `div` y)

divlist :: Integral a => [a] -> [a] -> [Maybe a]
divlist = zipWith safeDiv
```

Řeš. 4.2.3

```
mayZip :: [a] -> [b] -> [(Maybe a, Maybe b)]
mayZip (a : xa) (b : xb) = (Just a, Just b) : (mayZip xa xb)
mayZip []       (b : xb) = (Nothing, Just b) : (mayZip [] xb)
mayZip (a : xa) []       = (Just a, Nothing) : (mayZip xa [])
mayZip []       []       = []
```

Řeš. 4.3.1

a) `Zero`, `Succ Zero`, `Succ (Succ Zero)`, `Succ (Succ (Succ Zero))`, ...

b) Zajistí, že kompilátor deklaruje `Nat` jako instanci typové třídy `Show` (tj. typové třídy poskytující funkci `show`, která umožní převést hodnotu typu na jeho řetězcovou interpretaci) a na základě definice datového typu `Nat` automaticky definuje intuitivním způsobem funkci `show`, tj. např. `show (Succ (Succ Zero))` ⇝* `"Succ (Succ Zero)"`.

c) Využijeme například analogii s Peanovými čísly – přirozenými čísly definovanými pomocí nuly a funkce následníka. Hodnotový konstruktor `Zero` odpovídá nule, budete tedy psát `"0"`. Hodnotový konstruktor `Succ` pak představuje přičtení jedničky, budeme tedy psát `"1+"`. Definice instance pak může vypadat třeba následovně:

```
data Nat' = Zero' | Succ' Nat'
instance Show Nat' where
    show Zero'     = "0"
    show (Succ' x) = "1+" ++ show x
```

Poznamenejme ještě, že pokud definujeme svoji vlastní instanci, klauzuli `deriving Show` musíme z definice typu odstranit.

d)
```
natToInt :: Nat -> Int
natToInt Zero     = 0
natToInt (Succ x) = 1 + natToInt x
```

e) Takováto hodnota má tvar `Succ (Succ (Succ (Succ ...)))`. Lze se tedy inspirovat například funkcí `repeat`:

```
natInfinity :: Nat
natInfinity = Succ natInfinity
```

Řeš. 4.3.2

a)
```
Con 3.14
```

b)
```
eval :: Expr -> Float
eval (Con x)   = x
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x - eval y
eval (Mul x y) = eval x * eval y
eval (Div x y) = eval x / eval y
```

c) Pro získání hodnoty z výrazu tvaru `Just x` použijeme standardně definovanou funkci

```
fromJust :: Maybe a -> a
fromJust (Just x) = x

evaluate :: Expr -> Maybe Float
evaluate (Con x) = Just x
evaluate (Add x y) = if evx /= Nothing && evy /= Nothing
    then Just (fromJust evx + fromJust evy)
    else Nothing
    where
        evx = evaluate x
        evy = evaluate y

-- vyhodnocovani pro konstruktory Sub a Mul je uplne analogicke
-- vyhodnocovani Add, proto ho neuvadime

evaluate (Div x y) = if evx /= Nothing && evy /= Nothing
    then if fromJust evy == 0
        then Nothing
        else Just (fromJust evx / fromJust evy)
```

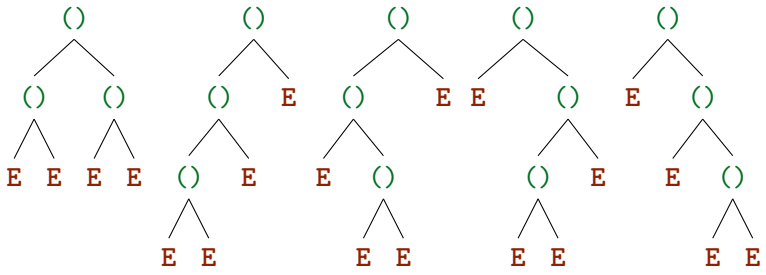
```
        else Nothing
        where
            evx = evaluate x
            evy = evaluate y
```

Řeš. 4.3.3

```
data Expr = Con Float     | Var
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr

eval' :: Float -> Expr -> Float
eval' _ (Con x)   = x
eval' v (Var)     = v
eval' v (Add x y) = eval' v x + eval' v y
eval' v (Sub x y) = eval' v x - eval' v y
eval' v (Mul x y) = eval' v x * eval' v y
eval' v (Div x y) = eval' v x / eval' v y
```

Řeš. 4.3.4 a) Jsou to tyto stromy:



```
tree1 = Node () (Node () Empty Empty) (Node () Empty Empty)
tree2 = Node () (Node () (Node () Empty Empty) Empty) Empty
tree3 = Node () (Node () Empty (Node () Empty Empty)) Empty
tree4 = Node () Empty (Node () (Node () Empty Empty) Empty)
tree5 = Node () Empty (Node () Empty (Node () Empty Empty))
```

b) Nechť $\#_{()}(n)$ je počet stromů typu `BinTree ()`. Pak lze nahlédnout, že

$$\#_{()}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{n-1} \#_{()}(i)\#_{()}(n-i-1) & \text{if } n > 0 \end{cases}$$

Pokud bychom chtěli znát konkrétní hodnoty, můžeme formuli přepsat

```
count 0 = 1
count n = sum $ map (\i -> count i * count (n - i - 1)) [0 .. n - 1]
```

čímž lehce zjistíme, že `map count [0..5]` $\rightsquigarrow$ `[1,1,2,5,14,42]`

c) Pro `BinTree Bool` platí

$$\#_{\texttt{Bool}}(n) = 2^n \#_{()}(n)$$

Obecně pro `BinTree` t máme:

$$\#_t(n) = |t|^n \#_{()}(n),$$

kde $|t|$ je počet různých hodnot typu t. Hledané hodnoty $\#_{\texttt{Bool}}(n)$ jsou:

```
countT t n = t ^ n * count n
map (countT 2) [0, 1, 2, 3, 4, 5] ⇝* [1, 2, 8, 40, 224, 1344]
```

Řeš. 4.3.5 a)
```
treeSize :: BinTree a -> Int
treeSize Empty          = 0
treeSize (Node _ t1 t2) = 1 + treeSize t1 + treeSize t2
```
b)
```
listTree :: BinTree a -> [a]
listTree Empty          = []
listTree (Node v t1 t2) = listTree t1 ++ [v] ++ listTree t2
```
c)
```
height :: BinTree a -> Int
height Empty        = 0
height (Node x l r) = 1 + max (height l) (height r)
```
d)
```
longestPath :: BinTree a -> [a]
longestPath Empty          = []
longestPath (Node v t1 t2) = if length p1 > length p2
    then v : p1
    else v : p2
    where
        p1 = longestPath t1
        p2 = longestPath t2
```

Řeš. 4.3.6 a)
```
fullTree :: Int -> a -> BinTree a
fullTree 0 _ = Empty
fullTree n v = Node v (fullTree (n - 1) v) (fullTree (n - 1) v)
```
b)
```
treeZip :: BinTree a -> BinTree b -> BinTree (a, b)
treeZip (Node x1 l1 r1) (Node x2 l2 r2) =
    Node (x1, x2) (treeZip l1 l2) (treeZip r1 r2)
treeZip _ _ = Empty
```

Řeš. 4.3.7
```
treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)
treeMayZip (Node a l1 r1) (Node b l2 r2) =
    Node (Just a, Just b) (treeMayZip l1 l2) (treeMayZip r1 r2)
treeMayZip (Node a l1 r1) Empty =
    Node (Just a, Nothing) (treeMayZip l1 Empty) (treeMayZip r1 Empty)
treeMayZip Empty (Node b l2 r2) =
    Node (Nothing, Just b) (treeMayZip Empty l2) (treeMayZip Empty r2)
treeMayZip Empty Empty = Empty
```

Řeš. 4.3.8
```
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.

Řeš. 4.3.9 a)
```
-- Rozšíření množiny hodnot typu `a` o nekonečno
data MayInf a = Inf | NegInf | Fin a
                deriving (Eq)

instance (Ord a) => Ord (MayInf a) where
    _       <= Inf     = True
    Inf     <= _       = False
```

```
        NegInf  <= _       = True
        _       <= NegInf  = False
        (Fin a) <= (Fin b) = a <= b

    isTreeBST :: (Ord a) => BinTree a -> Bool
    isTreeBST = isTreeBST' NegInf Inf

    isTreeBST' :: (Ord a) => MayInf a -> MayInf a -> BinTree a -> Bool
    isTreeBST' _   _    Empty        = True
    isTreeBST' low high (Node v l r) = let v' = Fin v in
        low <= Fin v && Fin v <= high &&
        isTreeBST' v' high r &&
        isTreeBST' low v' l

    -- alternativní definice
    isTreeBST2 :: (Ord a) => BinTree a -> Bool
    isTreeBST2 = isAscendingList . inorder

    inorder :: BinTree a -> [a]
    inorder Empty        = []
    inorder (Node v l r) = inorder l ++ [v] ++ inorder r

    isAscendingList :: (Ord a) => [a] -> Bool
    isAscendingList [] = True
    isAscendingList l  = and $ zipWith (<=) l (tail l)
```

b)
```
    searchBST :: (Ord a) => a -> BinTree a -> Bool
    searchBST _ Empty = False
    searchBST k (Node v l r) = case compare k v of
        EQ -> True
        LT -> searchBST k l
        GT -> searchBST k r
```

Řeš. 4.4.1
```
roseTreeSize :: RoseTree a -> Int
roseTreeSize (RoseNode _ subtrees) = 1 + sum (map roseTreeSize subtrees)

roseTreeSum :: Num a => RoseTree a -> a
roseTreeSum (RoseNode v subtrees) = v + sum (map roseTreeSum subtrees)

roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap f (RoseNode v subtrees) =
    RoseNode (f v) (map (roseTreeMap f) subtrees)
```

Řeš. 4.4.2
```
data Bit = O | I
           deriving Show

toBitString :: Int -> [Bit]
toBitString 0 = [O]
toBitString a = (if a `mod` 2 == 1 then I else O)
    : toBitString (a `div` 2)
```

```
fromBitString :: [Bit] -> Int
fromBitString (O : xs) = 2 * fromBitString xs
fromBitString (I : xs) = 1 + 2 * fromBitString xs
fromBitString []       = 0

insert :: IntSet -> Int -> IntSet
insert set num = insert' set (toBitString num)

insert' :: IntSet -> [Bit] -> IntSet
insert' SetLeaf []         = SetNode True SetLeaf SetLeaf
insert' (SetNode _ l r) [] = SetNode True l r
insert' SetLeaf (O : bits) = SetNode False (insert' SetLeaf bits) SetLeaf
insert' SetLeaf (I : bits) = SetNode False SetLeaf (insert' SetLeaf bits)
insert' (SetNode end l r) (O : bits) = SetNode end (insert' l bits) r
insert' (SetNode end l r) (I : bits) = SetNode end l (insert' r bits)

find :: IntSet -> Int -> Bool
find set num = find' set (toBitString num)

find' :: IntSet -> [Bit] -> Bool
find' (SetNode True _ _) []    = True
find' _ []                     = False
find' SetLeaf _                = False
find' (SetNode _ l _) (O : xs) = find' l xs
find' (SetNode _ _ r) (I : xs) = find' r xs

listSet :: IntSet -> [Int]
listSet set = listSet' set []

listSet' :: IntSet -> [Bit] -> [Int]
listSet' SetLeaf _                = []
listSet' (SetNode False l r) bits = listSet' l (O : bits)
    ++ listSet' r (I : bits)
listSet' (SetNode True l r) bits  = fromBitString (reverse bits)
    : listSet' l (O : bits) ++ listSet' r (I : bits)
```

Řeš. 4.4.3

```
data SeqSet a = SeqNode Bool [(a, SeqSet a)]
              deriving Show

son :: (Eq a) => a -> [(a, SeqSet a)] -> SeqSet a
son a anc = getSon $ lookup a anc
    where
        getSon Nothing = SeqNode False []
        getSon (Just node) = node

insertSeq :: Eq a => SeqSet a -> [a] -> SeqSet a
insertSeq (SeqNode _ anc)   []       = SeqNode True anc
insertSeq (SeqNode end anc) (a : xa) = let s = son a anc in
    SeqNode end ((a, insertSeq s xa) : filter (not . (a==) . fst) anc)
```

```
findSeq :: Eq a => SeqSet a -> [a] -> Bool
findSeq (SeqNode end anc) []       = end
findSeq (SeqNode end anc) (a : xa) = findSeq (son a anc) xa
```

Cvičení 5: Lenost, intensionální seznamy, foldy

Řeš. 5.1.1 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 Fešák 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).

Řeš. 5.1.2

```
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[6]. 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. 5.1.4

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

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

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. 5.1.5
```
addNumbers :: [String] -> [String]
addNumbers ns = zipWith (\i n -> show i ++ ". " ++ n) [1 ..] ns
```

Řeš. 5.1.6 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. 5.1.8

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. 5.1.9 Existuje více řešení. Označíme je postupně `fib`N.

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

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

-- efektivni seznamova definice
```

```
fib3 = fib' (0, 1)
    where
        fib' (x, y) = x : fib' (y, x + y)

-- efektivni definice funkce s akumulacnim parametrem, odvozena z fib3
fib4 n = fib' n (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. 5.1.10 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. 5.1.11

```
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. 5.1.12

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

Řeš. 5.1.13

```
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. 5.1.14
```
infFinTree = Node "strom" infFinTree (Node "konec" Empty Empty)
```

Řeš. 5.2.1

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

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

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

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

Řeš. 5.2.2
```
allPairs people = [ (fst m, fst f) | m <- people, not (snd m),
                                     f <- people, snd f ]
```

Jiným funkčním řešením by bylo například:

```
allPairs people = [ (fst m, fst f) | m <- people, f <- people,
                                     not (snd m), snd f ]
```

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, aby se nevyhovující přiřazení vyloučila co nejdříve. Vzájemné pořadí generátorů pak ovlivňuje, zda jsou ve výsledném seznamu shlukovány páry podle mužů, nebo žen.

Řeš. 5.2.3
```
divisors :: Int -> [Int]
divisors n = [ x | x <- [1 .. n], mod n x == 0 ]
```

Řeš. 5.2.4

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. 5.2.5

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

Řeš. 5.2.6

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 s =
    [m : t | (m, n) <- zip s . tails . tail $ s, t <- comb (k - 1) n]
        where
            tails []       = [[]]
            tails (x : s) = (x : s) : tails s
```

Tady lze případně použít funkci `tails` z modulu `Data.List`, viz http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-List.html#v:tails.

Řeš. 5.2.7 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. 5.2.8
```
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.

Řeš. 5.2.9
```
threeSum = [ (x, y, z) | z <- [2 ..], x <- [1 .. z - 1], let y = z - x ]
```

Jak bylo nastíněno v řešení , 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. 5.2.10
```
nonNegativePairs = [ (x, y) | z <- [2 ..], x <- [1 .. z - 1], let y = z -
↪  x ]
```

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

Řeš. 5.2.11 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
productFold' :: Num a => [a] -> a
productFold' = foldr (\e t -> e * t) 1
```

c)
```
orFold :: [Bool] -> Bool
orFold = foldr (||) False
orFold' :: [Bool] -> Bool
orFold' = foldr (\e t -> e || t) False
```

d)
```
andFold :: [Bool] -> Bool
andFold = foldr (&&) True
andFold' :: [Bool] -> Bool
andFold' = foldr (\e t -> e && t) True
```

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

f)
```
minimumFold :: Ord a => [a] -> a
minimumFold = foldr1 min
minimumFold' :: Ord a => [a] -> a
minimumFold' = foldr1 (\e t -> if e < t then e else t)
minimumFold'' :: Ord a => [a] -> a
minimumFold'' list = foldr min (head list) list
```

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

```
maximumFold' :: Ord a => [a] -> a
maximumFold' = foldr1 (\e t -> if e < t then t else e)
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) Nejsnáze se význam tohoto výrazu objasní na příkladě. Na základě typu λ-abstrakce vidíme, že funkce pracuje na číslech.

```
foldr1 (\x s -> x + 10 * s) [1, 2, 3] ⇝* 1 + 10 * (2 + 10 * 3)
    ⇝* 321
```

b) Tento případ je podobný jako u výrazu s `foldr1`. Opět se podívejme na příklad vyhodnocení:

```
foldl1 (\s x -> 10 * s + x) [1, 2, 3] ⇝* 10 * (10 * 1 + 2) + 3
    ⇝* 123
```

Tedy funkce převede seznam čísel reprezentující dekadický rozklad čísla na jedno číslo.

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) Při určování typu lze postupovat následovně 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)
```
composeFold :: [(a -> a)] -> a -> a
composeFold = foldr (.) id
composeFold' :: [(a -> a)] -> a -> a
composeFold' = flip (foldr id)
```

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

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

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

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

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

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 list = foldr (\e (tMin, tMax) -> (min e tMin, max e tMax))
                        (head list, head list) list
```

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 zbytek seznamu od $(n+1)$-tého prvku. Pokud bychom předpokládali, že tato funkce existuje, musela by fungovat korektně i na dvojici seznamů `[1, 2, 3]` a `[0, 1, 2, 3]`. Avšak v jistém okamžiku bude volána tak, že dostane jako argument hodnotu `1` a výsledek výrazu `foldr f [] [2, 3]`. Pak ale nelze rozeznat, o který případ se jedná, přičemž v jednom má vrátit `[1, 3]` a v druhém `[0, 2]`.

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 xs)
```

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
```

Cvičení 6: Vstup a výstup

Řeš. 6.1.1 a)
```
f  = getLine >>= \s -> putStrLn s
f' = getLine >>= putStrLn
```

b)
```
f = getLine >>= putStrLn . reverse
```

c)
```
f = getLine >>= \s -> putStrLn $ if null s then "<empty>" else s
```

d)
```
f = getLine >>= \s -> putStrLn s >> pure s
```

Řeš. 6.1.2
```
getInteger :: IO Integer
getInteger = getLine >>= \num -> pure (read num :: Integer)
```

Řeš. 6.1.3
```
loopecho = getLine >>= \s ->
           if null s then pure ()
                     else putStrLn s >>
                          loopecho
```

Řeš. 6.1.4
```
import Data.Char
getSanitized :: IO String
getSanitized = getLine >>= pure . filter isAlpha
```

Řeš. 6.1.5
```
import System.Directory

checkFile :: IO ()
checkFile = putStr "File: " >> getLine >>=
```

```
        doesFileExist >>= \b ->
        putStrLn $ if b then "File exists." else "No such file."
```

Řeš. 6.1.6
```
x >> f = x >>= \_ -> f
```

Řeš. 6.1.7
```
runLeft = foldl (\a b -> a >> b) (pure ())
runLeft' = foldr (\a b -> a >> b) (pure ())
runRight = foldl (\a b -> b >> a) (pure ())
runRight' = foldr (\a b -> b >> a) (pure ())
```

Všimněte si, že nezáleží na výběru akumulační funkce. Aplikací funkcí `foldl` a `foldr` sice vzniknou jiné stromy aplikací operátorů `>>`, ale pořadí listů zůstává stejné a operace `>>` je asociativní. Pro lepší názornost si nakreslete obrázek příslušných katamorfizmů.

Řeš. 6.2.2
```
leftPadTwo :: IO ()
leftPadTwo = do
    str1 <- getLine
    let l1 = length str1
    str2 <- getLine
    let l2 = length str2
    let m = max l1 l2
    putStrLn (replicate (m-l1) ' ' ++ str1)
    putStrLn (replicate (m-l2) ' ' ++ str2)
```

Řeš. 6.2.3
```
main = getLine >>= \f ->
       getLine >>= \s ->
       appendFile f (s ++ "\n")
```

Řeš. 6.2.4
```
query' :: String -> IO Bool
query' question =
    putStrLn question >> getLine >>= \answer -> pure (answer == "ano")
```

Nebo lze upravit vzniklou λ-abstrakci na pointfree tvar:

```
query'' :: String -> IO Bool
query'' question =
    putStrLn question >> getLine >>= pure . (== "ano")
```

Řeš. 6.2.5 a)
```
query2 :: String -> IO Bool
query2 question = do
    putStrLn question
    answer <- getLine
    if answer == "ano"
        then pure True
        else if answer == "ne"
            then pure False
            else query2 question
```

b)
```
import Data.Char

query3 :: String -> IO Bool
query3 question = do
    putStrLn question
    answer <- getLine
```

```
    let lcAnswer = map toLower answer
    if lcAnswer `elem` ["ano", "áno", "yes"] then
        pure True
    else
        if lcAnswer `elem` ["ne", "nie", "no"] then
            pure False
        else
            query3 question
```

Řeš. 6.2.6

a) Typ `IO String`; akce, která při spuštění načte řádek vstupu, jenž se stane vnitřním výsledkem.

b) Není výrazem. Při zadání do interpretu nebo do souboru se takto zadefinuje nová akce `x :: IO String`, která se chová stejně jako `getLine`.

c) Ekvivalentní `getLine`; `x` je lokální akce zadefinovaná jako výše.

d) Syntakticky nesprávné; konstrukci `<-` je lze použít pouze v `do`-bloku nebo intensionálním zápisu seznamu.

e) Ekvivalentní `getLine`; `x :: String` představuje načtený řádek.

f) Typově nesprávné; na pravé straně `>>=` má v tomto případě stát funkce typu `String -> IO a`.

g) Může to být překvapivé, ale výraz je správně utvořený. Má ale poněkud děsivý typ `Monad m => m (IO String)`, což pro účely tohoto cvičení můžeme číst jako `IO (IO String)`. Je to akce, jejímž vnitřním výsledkem je po spuštění akce `getLine`.

h) Ekvivalentní `getLine`; `x` je opět lokálně zadefinovaná akce.

i) Ekvivalentní `getLine`; `x :: String` představuje načtený řádek. Analogické k e).

j) Typově nesprávné; poslední výraz `do`-bloku musí být typu „akce“. Analogické k f).

Řeš. 6.3.1 Do souboru s akcí `leftPadTwo`, nechť se jmenuje třeba `Cv06.hs`, přidáme:

```
main :: IO ()
main = putStrLn "Always two lines there are:" >> leftPadTwo
```

Soubor přeložíme a spustíme:

```
$ ghc Cv06.hs
[1 of 1] Compiling Main             ( Cv06.hs, Cv06.o )
Linking Cv06 ...
$ ./Cv06
Always two lines there are:
They're taking the hobbits to Isengard!
Tell me where is Gandalf, for I much desire to speak with him.
                        They're taking the hobbits to Isengard!
Tell me where is Gandalf, for I much desire to speak with him.
$
```

Řeš. 6.3.2

```
biggest :: Integer -> Integer -> Integer -> Integer
biggest x y z = max (max x y) z

lowerTwo :: Integer -> Integer -> Integer -> Integer
lowerTwo x y z = x + y + z - biggest x y z
```

```
getInteger :: IO Integer
getInteger = getLine >>= \num -> pure (read num :: Integer)

main :: IO ()
main = do
  putStrLn "Enter first number:"
  x <- getInteger
  putStrLn "Enter second number:"
  y <- getInteger
  putStrLn "Enter third number:"
  z <- getInteger
  if biggest x y z < lowerTwo x y z
      then putStrLn "ANO"
      else putStrLn "NE"
```

Řeš. 6.3.3

```
import System.Directory

main :: IO ()
main = do putStr "Enter filename: "
          fileName <- getLine
          fileExists <- doesFileExist fileName
          if fileExists then do
              fileContents <- readFile fileName
              putStr fileContents
          else putStrLn "Impossible. Perhaps the archives are incomplete."
```

Řeš. 6.3.4 Obecně, rekurze v kontextu `IO` umožňuje opakované vykonávání akcí. V tomto případě vidíme, že od první akce, která je součástí `main'`, až po její další výskyt se opakuje načtení řetězce a výpis jeho převrácené podoby. Dobře použitá rekurze musí být vhodným způsobem ukončitelná. Význam kódu měl nejspíš být takový, že zadáme-li prázdný řetězec, dojde k ukončení rekurze a akce se nebude znovu opakovat. Autor zde však rekurzi neuhlídal a `main'` se volá pokaždé; akci tedy není jak ukončit. Náprava je jednoduchá: stačí poslední řádek odsadit tak, aby byl součástí vnořeného `do`-bloku.

V takto jednoduchém případě se místo vnořeného `do`-bloku dá bez ztráty čitelnosti použít `>>`:

```
main :: IO ()
main = do putStr "Enter string: "
          s <- getLine
          if null s then putStrLn "You shall not pass!"
          else putStrLn (reverse s) >> main
```

Řeš. 6.3.5

```
import Text.Read

requestInteger :: String -> IO (Maybe Integer)
requestInteger delim = do
    str <- getLine
    if str == delim then pure Nothing else f (readMaybe str)
    where
        f Nothing = requestInteger delim
        f mayInt  = pure mayInt
```

Řeš. 6.3.7

```
leftPad :: IO ()
leftPad = leftPad' []

leftPad' ::  [String] -> IO ()
leftPad' lines = do
    line <- getLine
    if line == "."
        then printLines (maximum (map length lines)) (reverse lines)
        else leftPad' (line:lines)

printLines :: Int -> [String] -> IO ()
printLines _ [] = pure ()
printLines maxLen (line:xs) = do
    let prefix = replicate (maxLen - length line) ' '
    putStrLn (prefix ++ line)
    printLines maxLen xs
```

Řeš. 6.3.10 Dvojice obsahující v obou složkách týž řádek vstupu. Začne se vyhodnocovat první složka dvojice; vyhodnocení `getLine'` způsobí přečtení řádku vstupu, který se stane výsledkem výrazu. Kvůli línému vyhodnocování se už ve druhé složce nemá co vyhodnocovat (a tedy se nenačte další řádek), protože výraz `getLine'` už vyhodnocený je.

Příklad ilustruje jeden z důvodů, proč je v Haskellu potřeba řešit vstup a výstup poněkud neintuitivním způsobem. Dalším důvodem je pořadí spuštění – u vstupně-výstupních akcí ho většinou chceme přesně stanovené. To je u normální redukční strategie obtížnější dosáhnout. Řekněme, že výrazy s vedlejšími efekty se vždy vyhodnocují znovu (aby vůbec došlo k těm efektům). Například vyhodnocení `getLine'` vždy způsobí, že se načte řádek. Vezměme si hypotetickou funkci `writeFile' :: FilePath -> String -> ()`. Budeme chtít napsat program, který načte název souboru a obsah, který do něj uloží. Které řešení je vezme v jakém pořadí?

```
main =      writeFile' getLine' getLine'
main = flip writeFile' getLine' getLine'
```

Odpověď zní: nevíme, ale v obou bude stejné. Záleží na tom, co `writeFile'` bude chtít načíst první (dá se očekávat, že to bude název souboru). Zároveň s tím `flip` nic neudělá, protože oba argumenty jsou nevyhodnocený výraz (s vedlejšími efekty), a byť už máme zajištěno, že se oba výskyty vyhodnotí zvlášť, redukční strategie stále nevynucuje, aby se to událo před voláním `writeFile'`.

Chceme-li vynutit pořadí vykonávání akcí, můžeme mezi nimi uměle zavést závislost. Například první funkce by vracela kromě načteného řádku i nějaké nahodilé číslo, které by se jako štafetový kolík předalo druhé funkci. Ta by tak mohla byt vyhodnocena až poté, co by došlo k vyhodnocení první. Typ by se přitom změnil na `getLine'' :: Integer -> (String, Integer)`. Implementace řetězení funkcí je jistě zřejmá. A protože k takovému řetězení by docházelo často, vymysleli bychom si na něj nějaký hezký operátor a nazvali jej třeba `>>`.

Článek `https://wiki.haskell.org/IO_inside` názorně krok po kroku ukazuje, že myšlenka štafetového kolíku je už poměrně blízko skutečné implementaci `IO`. Všem nebojácným zájemcům o Haskell ho Pan Fešák doporučuje si přečíst.

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.

04_trees.hs

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

tree01 :: BinTree Int
tree01 = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
                (Node 6 (Node 5 Empty Empty) (Node 7 Empty Empty))

tree02 :: BinTree Int
tree02 = Node 9 Empty (Node 10 Empty (Node 11 Empty (Node 12 Empty Empty)))

tree03 :: BinTree Int
tree03 = Node 8 tree01 tree02

tree04 :: BinTree Int
tree04 = Node 4 (Node 2 Empty (Node 3 Empty Empty))
                (Node 6 (Node 5 Empty Empty) Empty)

tree05 :: BinTree Int
tree05 = Node 100 (Node 101 Empty
                            (Node 102 (Node 103 Empty
                                                (Node 104 Empty Empty))
                                      Empty))
                  (Node 99 (Node 98 Empty Empty)
                           (Node 98 Empty Empty))
```

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
```

```haskell
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))
```

06_guess.hs

```haskell
import Data.Char

main = guess 1 10

query ot = do putStr ot
              ans <- getLine
              pure (ans == "ano")

guess :: Int -> Int -> IO ()
guess m n = do putStrLn ("Mysli si cele cislo od " ++ show m
                         ++ " do " ++ show n ++ ".")
               kv m n

kv m n = do
     if m == n
     then putStrLn ("Je to " ++ show m ++ ".")
     else do o <- query $ "Je tve cislo vetsi nez " ++ show k ++ "? "
             if o then kv (k + 1) n else kv m k
  where k = (m + n) `div` 2
```