Cvičení 3: Funkce vyšších řádů, λ-funkce, částečná aplikace, skládání

Před třetím cvičením je zapotřebí znát:

- lokální definice pomocných funkcí pomocí klauzule `where`

```
power :: Int -> Int
power x = result
  where result = x * x
```

- 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, tj. například `\x y -> x + y`
- co je částečná aplikace;
- použití operátoru `(.) :: (b -> c) -> (a -> b) -> (a -> c)` pro skládání unárních funkcí;

Etudy

Etuda 3.η.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]`?

Pan Fešák připomíná: Lambda je řecké písmeno (λ; velká lambda je Λ). V teorii programování se používá pro uvození anonymní funkce a proto se těmto funkcím říká lambda funkce. Protože písmeno λ není na běžných klávesnicích, používá se místo něj v Haskellu `\`.

Etuda 3.η.2 Naprogramujte funkci `oddify :: Integral a => [a] -> [a]`, která projde vstupní seznam čísel a čísla, která nejsou lichá, přemění na lichá. Použijte funkci `map`, nebo `filter`. Může se vám také hodit lambda funkce.

Například:

```
oddify [1, 2, 3, 4] ⇝* [1, 3, 3, 5]
```

Etuda 3.η.3 S využitím funkce `oddify` z předchozího příkladu 3.η.2 naprogramujte funkci `inputWithOddified :: Integral a => [a] -> [(a, a)]`, která pro vstupní seznam čísel vrátí seznam dvojic, kde první složka odpovídá prvku ze vstupního seznamu a druhá složka výstupu funkce `oddify`. Například:

```
inputWithOddified [1, 2, 3, 4] ⇝* [(1, 1), (2, 3), (3, 3), (4,
↪ 5)]
```

Etuda 3.η.4 Bez použití interpretu určete hodnoty následujících výrazů:

a) `(head . head) [[1,2,3], [4, 5, 6]]`
b) `((\(x:xs) -> xs) . head) "ahoj"`
c) `let g x = x : [x] in g 10`
d) `(last . last) [[1, 2], [3, 4], []]`
e) `let f g x = (g . g) x in f (+ 21) 0`

Pokud výraz nelze vyhodnotit, určete přesně, která část výpočtu selhává a proč.

Etuda 3.η.5 Vysvětlete rozdíl mezi následujícími dvojicemi výrazů:

a) `(f . g) x` versus `g (f x)`
b) `(f . g) x` versus `f (g x)`
c) `(* 2) . (+ 2)` versus `\x -> (x + 2) * 2`
d) `head . head [[1], [2], [3]]` versus `(head . head) [[1], [2], [3]]`

> Doposud jsme se se bavili o n-árních funkcích, tedy o funkcích, které braly na vstupu n argumentů a vracely nějakou hodnotu. Ve skutečnosti ale v Haskellu nemáme funkce, které by braly více než jeden argument – jinak řečeno, funkce v Haskellu jsou nejvýše unární. Jak je to možné?
>
> Kouzlo tkví v jiném pohledu na funkce jako takové. Místo toho, abychom se na funkci dívali jako na něco, co bere n argumentů a vrátí hodnotu, se na ni můžeme dívat jako na unární funkci, která vrátí jinou $(n-1)$-ární funkci. Aritu potom můžeme brát jako počet unárních funkcí, které dostaneme postupnou aplikací na argumenty.
>
> Vezměme si například funkci
>
> ```
> add :: Int -> Int -> Int
> add x y = x + y
> ```
>
> Doposud bychom řekli, že funkce `add` bere dva argumenty a ty sečte.
>
> Pokud se na ni ale podíváme druhým způsobem, zjistíme, že typ funkce `add` lze ekvivalentně zapsat jako `add :: Int -> (Int -> Int)`, a tedy že bere jeden argument – `x` – a vrací funkci, která ke svému vstupu přičte dané `x`.
>
> A přesně tuto myšlenku využíváme u částečné aplikace. Díky tomu, že funkci dáme jeden argument, zafixujeme jeho hodnotu a dostaneme funkci, která *nasytí* zbytek. Potom pro nás není problém zavést si třeba funkci `addTo42` jako
>
> ```
> addTo42 :: Int -> Int
> addTo42 = add 42
> ```
>
> Tato funkce svůj vstupní argument přičte k hodnotě `42`. Na tento zápis se můžeme dívat i tak, že do `addTo42` uložíme funkci, kterou nám vrátí výraz `add 42`.
>
> Protože druhý pohled na funkce je Haskellu vlastní a operátory nejsou nic jiného než funkce, mohli jsme například úlohu 3.ψ.2 vyřešit jako `prepend42 = (:) 42`.

Etuda 3.η.6 Vysvětlete, co dělají následující funkce, a najděte argumenty, na něž je lze aplikovat. Následně si chování ověřte v GHCi.

a) `take 4`
b) `(++) "Hello, "`
c) `zip3 [1, 2, 3] ["a", "b"]`
d) `(^ 2)`

Etuda 3.η.7 Do následujících výrazů doplňte všechny implicitní závorky vycházející z částečné aplikace funkcí. Jinak řečeno, explicitně závorkami ukažte pořadí postupného vyhodnocování částečné aplikace.

a) `(==) 42 16`
b) `map ((==) 42) [1, 2, 3]`
c) `(g 4, f g 5)`
d) `zipWith3 f (g 4 a) xs`
e) typ: `Bool -> Bool -> Bool`
f) typ: `(a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]`
g) typ: `(b -> c) -> (a -> b) -> a -> c`

3.1 Užitečné seznamové funkce & lambda funkce

> **Pan Fešák připomíná** 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. 3.1.1 Naprogramujte funkci `filterOutShorter :: [String] -> Int -> [String]`, která pro vstupní seznam řetězců a zadané číslo `n`, vrátí seznam řetězců obsahující pouze ty, které mají délku alespoň `n`. Využijte knihovní funkci `filter` a vhodnou lambda funkci.

Př. 3.1.2 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 8 bodů,
c) funkci `successfulNames :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere jména studentů, kteří mají alespoň 8 bodů,
d) funkci `successfulStrings :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere studenty, kteří mají alespoň 8 bodů, a vrátí seznam řetězců ve tvaru `"jmeno: xxx b"` (*nápověda*: pro převod čísla na řetězec můžete použít funkci `show`).

Tedy například pro databázi

```haskell
st :: [(String, Integer)]
st = [("Finn", 5), ("Jake", 9), ("Bubblegum", 12),
      ("Ice King", 2), ("BMO", 15), ("Marceline", 9)]
```

budou požadované funkce vracet následující hodnoty:

```
getNames st ⇝* ["Finn", "Jake", "Bubblegum", "Ice King", "BMO",
                "Marceline"]
successfulRecords st ⇝* [("Jake", 9), ("Bubblegum", 12), ("BMO",
↪ 15),
                          ("Marceline", 9)]
successfulNames st ⇝* ["Jake", "Bubblegum", "BMO", "Marceline"]
successfulStrings st ⇝* ["Jake: 9 b", "Bubblegum: 12 b", "BMO:
↪ 15 b",
                          "Marceline: 9 b"]
```

Pan Fešák doporučuje: Pro práci s dvojicemi se můžou hodit funkce `fst` a `snd`, které vrací první, respektive druhou položku dvojice.

Pokud chceme s dvojicemi pracovat v lambda funkcích (nebo i pojmenovaných funkcích), tak dává často smysl používat vzory pro dvojice.

Př. 3.1.3 3.1.mf Které z funkcí z příkladu 3.ψ.1 lze elegantně naprogramovat pomocí funkce `map`? Které lze elegantně naprogramovat pomocí funkce `filter`? Všechny tyto funkce pomocí `map` a `filter` naprogramujte.

Př. 3.1.4 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. 3.1.5 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. 3.1.6 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. 3.1.7 Mějme výsledky běžeckého závodu reprezentované pomocí seznamu typu `[String]`, který obsahuje jména běžců seřazených od nejlepšího po nejhoršího, a seznam peněžních výher typu `[Integer]` (rovněž seřazených). 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. 3.1.8 Nahraďte v následujících výrazech `_lf1`/`_lf2` vhodnými lambda funkcemi tak, aby výsledek obou výrazů odpovídal uvedenému vyhodnocení. Jaká je arita vašich funkcí a jaké argumenty berou?

Poznámka: `zip3` a `zipWith3` jsou obdoby funkcí `zip` a `zipWith` které ale pracují se třemi seznamy místo dvou.

```
map _lf1 (zip3 [True, False, False, True, False]
               [1, 2, 3, 4] [16, 42, 7, 1, 666])
    ⇝* [1, 42, 7, 4]
zipWith3 _lf2 [7, 4, 11, 2] [5, 7, 1] [16, 5, 0, 1]
```

```
    ⇝* [16, 7, 11]
```

Př. 3.1.9 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. 3.1.10 Napište funkci, která zjistí, jestli jsou v seznamu čísel některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

Př. 3.1.11 Implementujte funkce `myMap`, `myFilter` a `myZipWith`, které se budou chovat jako knihovní funkce `map`, `filter` a `zipWith`.

Př. 3.1.12 Uvažte funkci `anyEven :: [Integer] -> Bool`, která rozhodne, jestli je v seznamu čísel nějaké sudé číslo, a funkci `allEven :: [Integer] -> Bool`, která rozhodne, jestli jsou všechna čísla v seznamu sudá. Najděte ve standardní knihovně funkci nebo funkce, pomocí kterých lze funkce `anyEven` a `allEven` implementovat jednoduše bez explicitního použití rekurze.

Př. 3.1.13 Zjistěte, co dělají funkce `takeWhile` a `dropWhile`.

Př. 3.1.14 Slovně popište, co dělají následující funkce, určete jejich arity a typy:

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

Př. 3.1.15 Mějme seznam barev ve formátu RGB reprezentovaném trojicí `(Int, Int, Int)`, kde první složka odpovídá červené, druhá zelené a třetí modré. Dále předpokládejme, že hodnoty v trojicích náleží do intervalu $[0, 255]$. Naprogramujte s využitím λ-funkcí

a) funkci `blueless :: [(Int, Int, Int)] -> [(Int, Int, Int)]`, která vrátí seznam barev obsahující pouze ty barvy, které neobsahují žádnou modrou složku,
b) funkci `greyscale :: [(Int, Int, Int)] -> [(Int, Int, Int)]`, která vrátí seznam těch barev, které jsou odstínem šedi (tj. všechny složky mají stejnou hodnotu),
c) funkci `polychromatic :: [(Int, Int, Int)] -> [(Int, Int, Int)]`, která vrátí seznam barev, které obsahují více než jednu nenulovou složku,
d) funkci `colorsToString :: [(Int, Int, Int)] -> [String]`, která převede barvy na řetězce ve formátu `"r: <Int> g: <Int> b: <Int>"`.

Př. 3.1.16 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.

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) `(^) 3`
b) `(^ 3)`
c) `(3 ^)`
d) `(- 2)`
e) `(2 -)`
f) `zipWith (+) [1, 2, 3]`
g) `map (++ "!")`
h) `/ 2`

Př. 3.2.2 3.3.part

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

a)

```
sumLists :: Num a => [a] -> [a] -> [a]
sumLists xs ys = zipWith (\x y -> x + y) xs ys
```

b)

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

c)

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

d)

```
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`

Př. 3.2.4 Rozhodněte, které z následujících výrazů jsou vzájemně ekvivalentní. Určete minimální arity všech identifikátorů v každém výrazu (tedy určete, jakou minimálně aritu daná entita musí mít, aby mohl být výraz korektní).

a) `a f g b`
b) `(a f) (g b)`
c) `((a f) g) b`
d) `a (f (g b))`
e) `(a f) g b`
f) `a f (g b)`
g) `a (f g b)`

Pan Fešák doporučuje: Všimněte si, že na to, abychom určili ekvivalenci výrazů, nemusíme vůbec vědět, co funkce dělají ani jaké jsou jejich typy. Podstatné je jen to, jak funguje volání funkcí v Haskellu. Z výrazu samotného můžeme vždy určit co jsou funkce a co jejich argumenty, přičemž argument samozřejmě může být funkčního typu, ale to nijak neovlivňuje volání funkce, do které ho předáváme (ovlivňuje to ale její typ).

Př. 3.2.5 Určete, které z následujících typů jsou vzájemně ekvivalentní. Určete aritu funkcí, které nesou tento typ a určete, zda jde o funkce vyšších řádů (tj. funkce, které berou jako argumenty funkce). Předpokládejme, že typy `A` až `E` jsou libovolné konkrétní typy.

a) `A -> B -> C -> D -> E`
b) `(A -> B) -> C -> D -> E`
c) `A -> (B -> C -> D -> E)`
d) `A -> B -> C -> (D -> E)`
e) `A -> ((B -> C) -> D -> E))`
f) `A -> (B -> (C -> (D -> E)))`
g) `(((A -> B) -> C) -> D) -> E`
h) `A -> (B -> C) -> D -> E`

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 >) . last)`
d) `const const`

Př. 3.3.2 Implementujte následující funkce s použitím `map`/`filter` a bez použití lambda funkcí a vlastních pomocných funkcí – tedy použijte vhodně částečnou aplikaci a skládání funkcí.

Pracovat budeme opět se záznamy o studentech z příkladu 3.1.2, tedy s typem

`[(String, Integer)]`.

a) `countStudentsByPoints :: Integer -> [(String, Integer)] -> Int`, která spočte, kolik studentů dostalo právě počet bodů daný druhým argumentem.

b) `studentNamesByPoints :: Integer -> [(String, Integer)] -> [String]`, která vrátí seznam jmen studentů, kteří dostali daný počet bodů.

c) `studentsStartingWith :: Char -> [(String, Integer)] -> [(String, Integer)]`, která vrátí seznam záznamů studentů, jejichž jméno začíná písmenem daným druhým argumentem.

```
countStudentsByPoints 5 st ⤳* 1
countStudentsByPoints 9 st ⤳* 2
countStudentsByPoints 0 st ⤳* 0

studentNamesByPoints 5 st ⤳* ["Finn"]
studentNamesByPoints 9 st ⤳* ["Jake", "Marceline"]

studentsStartingWith 'J' st ⤳* [("Jake", 9)]
studentsStartingWith 'B' st ⤳* [("Bubblegum", 12), ("BMO", 15)]
studentsStartingWith 'X' st ⤳* []
```

Př. 3.3.3 3.4.comp

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. 3.3.4 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.

```
flip (>) 42 . flip (*) 2

flip > 42 . flip * 2                flip (> 42) . flip (* 2)

(\x -> x > 42) . (* 2)              (>) 42 . (*) 2

(<) 42 . (*) 2                      \x -> (x * 2) > 42

(> 42) . (* 2)                      (* 2) . (> 42)

* 2 . > 42                          (> 42) (* 2)

\x -> ((> 42) . (* 2)) x
```

Př. 3.3.5 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. 3.3.6 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.

```haskell
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. 3.3.7 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. 3.3.8 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. 3.3.9 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. 3.3.10 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. 3.3.11 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. 3.3.12 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 třetího cvičení byste měli umět:

- 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 lambda funkce, a umět je použít;
- použít a otypovat částečně aplikovanou funkci;
- použít a otypovat operátorové sekce;
- skládat unární funkce pomocí operátoru `(.)`.

Psílohy

> Následující příklady jsou z 2. kapitoly, kde najdete i jejich řešení.

Př. 3.ψ.1 (Zpět: 3.1.3) Napište následující funkce pracující se seznamy čísel pomocí rekurze a vzorů:

a) `listSum :: Num n => [n] -> n`, která dostane seznam čísel a vrátí součet všech jeho prvků,
b) `oddLength :: [a] -> Bool`, která vrátí `True`, pokud je seznam liché délky, jinak `False` (bez použití funkce `length`),
c) `add1 :: Num n => [n] -> [n]`, která každé číslo ve vstupním seznamu zvýší o 1,
d) `multiplyN :: Num n => n -> [n] -> [n]`, která každé číslo ve vstupním seznamu vynásobí prvním argumentem funkce,
e) `deleteEven :: Integral i => [i] -> [i]`, která ze seznamu čísel odstraní všechna sudá čísla,
f) `deleteElem :: Eq a => a -> [a] -> [a]`, 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 :: Eq a => [a] -> [a] -> 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. 3.ψ.2 (Zpět: 3) Napište nerekurzivní funkci, která na začátek zadaného seznamu čísel vloží hodnotu `42`. Jaký má vaše funkce typ? Nezapomeňte na existenci typových tříd!

Řešení

Řeš. 3.η.2

```
oddify :: Integral a => [a] -> [a]
oddify xs = map (\x -> if odd x then x else x + 1) xs
```

Řeš. 3.η.3

```
inputWithOddified :: Integral a => [a] -> [(a, a)]
inputWithOddified xs = zip xs (oddify xs)
```

Řeš. 3.η.5

a) V prvním případě se aplikují funkce v opačném pořadí než v druhém.
b) Zápisy jsou ekvivalentní. V prvním případě bychom mohli vynechat formální parametr `x` i závorky.
c) Zápisy jsou ekvivalentní. *Poznámka*: Závorky kolem `x + 2` v druhém výrazu jsou nutné.
d) V prvním případě dostaneme chybu. Při vyhodnocování výrazu se nejdříve uzávorkují operandy `(.)`, a tudíž dostaneme výraz `head . (head [[1], [2], [3]])`. Protože `(head [[1], [2], [3]])` není unární funkcí, kterou po nás požaduje typ operátoru `(.)`, dostaneme typovou chybu. Druhý výraz je validní.

Řeš. 3.η.6

a) `(take 4) [1, 2, 3, 4, 5, 6, 7, 8]` $\rightsquigarrow^*$ `[1, 2, 3, 4]`
Funkce, která očekává seznam a zkrátí ho na nejvýše 4 prvky.

b) `((++) "Hello, ") "World"` $\rightsquigarrow^*$ `"Hello, World"`
Funkce, která očekává řetězec a přidá před něj řetězec `"Hello, "`.

c) `(zip3 [1, 2, 3] ["a", "b"]) [True]` $\rightsquigarrow^*$ `[(1, "a", True)]`
Funkce, která očekává jeden seznam a sezipuje ho se seznamy `[1, 2, 3]` a `["a", "b"]` na seznam trojic, přičemž prvky vstupního seznamu se dostanou do třetích složek trojic.

d) `(^ 2) 5` $\rightsquigarrow^*$ `25`
Funkce, která očekává číslo a spočítá jeho druhou mocninu.

Řeš. 3.η.7

a) `((==) 42) 16` – každá Haskellová funkce arity alespoň dva se nejprve aplikuje na svůj první argument, čímž vznikne funkce o jedna nižší arity, která se dále aplikuje na další argumenty.

b) `(map ((==) 42)) [1, 2, 3]` – stejný případ jako minulý bod: funkce `map` se nejprve částečně aplikuje na svůj první argument a výsledná funkce se aplikuje na druhý argument. Pokud bychom chtěli jít do detailu, měli bychom i rozepsat seznam pomocí `(:)` a `[]`: `(map ((==) 42)) (1:(2:(3:[])))`.

c) `(g 4, (f g) 5)` – je důležité si především všimnout, že `f` je (alespoň binární) funkce aplikovaná na dva argumenty `g` a `5`.

d) `( (zipWith3 f) ((g 4) a) ) xs` – všimněte si, že funkce stále není plně aplikovaná.

e) `Bool -> (Bool -> Bool)` – závorkování typu zprava přímo odpovídá závorkování výrazu zleva (porovnejte s `(||) False True`).

f) `(a -> b -> c -> d) -> ([a] -> ([b] -> ([c] -> [d])))` – všimněte si, že závorka končící před šipkou (`->`) má výrazně jiný význam, než ta začínající za šipkou.

g) `(b -> c) -> ((a -> b) -> (a -> c))` – všimněte si, že závorka kolem `a -> c` odpovídá obvyklému pohledu na funkci `(.)`, která má tento typ – tedy jako na binární funkci skládání funkcí (která produkuje funkci).

Řeš. 3.1.1

```
filterOutShorter :: [String] -> Int -> [String]
filterOutShorter ls n = filter (\str -> length str >= n) ls
```

Řeš. 3.1.2

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

successfulRecords :: [(String, Integer)] -> [(String, Integer)]
successfulRecords s = filter (\(_, p) -> p >= 8) s

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

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

Řeš. 3.1.3 Funkci `map` lze využít v případě, že potřebujeme jistým způsobem modifikovat každý prvek zadaného seznamu.

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

multiplyN' :: Integer -> [Integer] -> [Integer]
multiplyN' n xs = map (\x -> x * n) xs
```

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 (\x -> x /= n) xs
```

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

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

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

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. 3.1.4

```
import Data.Char
toUpperStr :: String -> String
```

```haskell
toUpperStr = map toUpper
```

Řeš. 3.1.5 Nejdřív si zadefinujeme pomocný predikát `isvowel`, který o znaku určí, jestli je samohláskou. Následně jednotlivé řetězce projdeme funkcí `filter`.

```haskell
vowels :: [String] -> [String]
vowels s = map (filter isvowel) s
  where
    isvowel :: Char -> Bool
    isvowel c = elem (toUpper c) "AEIOUY"
```

Řeš. 3.1.7

```haskell
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. 3.1.8

```haskell
map (\(x, y, z) -> if x then y else z)
    (zip3 [True, False, False, True, False]
          [1, 2, 3, 4] [16, 42, 7, 1, 666])
    ⇝* [1, 42, 7, 4]
zipWith3 (\x y z -> max x (max y z))
         [7, 4, 11, 2] [5, 7, 1] [16, 5, 0, 1]
    ⇝* [16, 7, 11]
```

V prvním případě se jedná o funkci arity 1, která bere trojici (kde první složka je `Bool` a další jsou čísla).

V druhém případě má lambda funkce aritu 3, bere tři čísla.

Řeš. 3.1.9

```haskell
neighbors :: [a] -> [(a, a)]
neighbors xs = zip xs (tail xs)
```

Řeš. 3.1.10

```haskell
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:

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

Řeš. 3.1.11

```haskell
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. 3.1.12 Využít můžeme funkce `any` a `all`.
http://haskell.fi.muni.cz/doc/base/Prelude.html#v:any
http://haskell.fi.muni.cz/doc/base/Prelude.html#v:all

Řeš. 3.1.13 http://haskell.fi.muni.cz/doc/base/Prelude.html#v:takeWhile
http://haskell.fi.muni.cz/doc/base/Prelude.html#v:dropWhile

Řeš. 3.1.15

```
blueless :: [(Int, Int, Int)] -> [(Int, Int, Int)]
blueless colors = filter (\(r, g, b) -> b == 0) colors

greyscale :: [(Int, Int, Int)] -> [(Int, Int, Int)]
greyscale colors = filter (\(r, g, b) -> r == g && g == b)
 ↪  colors

polychromatic :: [(Int, Int, Int)] -> [(Int, Int, Int)]
polychromatic cs = filter (\(r, g, b) -> (r > 0 && g > 0)
                                      || (g > 0 && b > 0)
                                      || (r > 0 && b > 0)) cs

colorsToString :: [(Int, Int, Int)] -> [String]
colorsToString cs = map (\(r, g, b) -> "r: " ++ show r ++ " g: "
                                    ++ show g ++ " b: " ++
                                     ↪  show b) cs
```

Řeš. 3.1.16

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

Řeš. 3.2.1

a) `(^) 3 2` ⇝* `9`
Funkce, která vypočítá danou mocninu trojky.

b) `(^ 3) 2` ⇝* `8`
Funkce, která dané číslo umocní na třetí.

c) `(3 ^) 2` ⇝* `9`
Funkce, která umocní trojku daným číslem.

d) `(- 2)` ⇝* `-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` ⇝* `42`.

e) `(2 -) 1` ⇝* `1`
Funkce, která dané číslo odečte od dvojky.

f) `(zipWith (+) [1, 2, 3]) [41, 14]` ⇝* `[42, 16]`

Funkce, která očekává seznam a po prvcích jej sečte se seznamem `[1, 2, 3]`, produkuje tedy seznam délky nejvýše 3.

g) `(map (++ "!")) ["ahoj", "hi"]` $\rightsquigarrow^*$ `["ahoj!", "hi!"]`
Funkce, která očekává seznam řecezů a produkuje nový seznam řetězců ve kterém je ke každému řetězci na konec přidán „!“.

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

Řeš. 3.2.2 a)

```
sumLists' :: Num a => [a] -> [a] -> [a]
sumLists' xs ys = zipWith (+) xs ys
```

b)

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

c)

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

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

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

d)

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

Řeš. 3.2.3

a) Ne. 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` $\equiv$ `f 1 g 2` $\equiv$ `(f 1) g 2` (tedy funkce `f` tu bere dva argumenty).

c) Ne, `(* 2) 3` $\equiv$ `(*) 3 2` $\equiv$ `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. 3.2.4 Následující výrazy jsou ekvivalentní:

- `a f g b` $\equiv$ `((a f) g) b` $\equiv$ `(a f) g b` (a $\equiv$ c $\equiv$ e)

  V těchto výrazech jsou arity následující:

  - `a` – arita alespoň 3 (bere minimálně 3 argumenty – `f`, `g`, `b`)
  - `f`, `g`, `b` – arita alespoň 0 (konstanta)

- `(a f) (g b)` $\equiv$ `a f (g b)` (b $\equiv$ f)

  Arity:

  - `a` – arita alespoň 2 (argumenty `f` a `(g b)`)
  - `g` – arita alespoň 1
  - `f`, `b` – arita alespoň 0

- `a (f (g b))` nemá ekvivalentní výraz (d)

  Arity:

  - `a`, `f`, `g` – arita alespoň 1

  - `b` – arita alespoň 0

- `a (f g b)` nemá ekvivalentní výraz (g)

  Arity:

  - `f` – arita alespoň 2
  - `a` – arita alespoň 1
  - `g`, `b` – arita alespoň 0

Řeš. 3.2.5 Následující typy jsou ekvivalentní:

- `A -> B -> C -> D -> E` ≡ `A -> (B -> C -> D -> E)` ≡ `A -> B -> C -> (D -> E)` ≡ `A -> (B -> (C -> (D -> E)))` (a ≡ c ≡ d ≡ f)

  Arita funkce je 4, nejedná se o funkci vyššího řádu.

- `(A -> B) -> C -> D -> E` nemá ekvivalentní typ (b)

  Arita je 3, jde o funkci vyššího řádu (1. argument je funkce `A -> B`).

- `A -> ((B -> C) -> D -> E)` ≡ `A -> (B -> C) -> D -> E` (e ≡ h)

  Arita 3, jde o funkci vyššího řádu (2. argument je funkce `B -> C`).

- `(((A -> B) -> C) -> D) -> E` nemá ekvivalentní typ (g)

  Arita 1, jde o funkci vyššího řádu (která bere funkci vyššího řádu).

Řeš. 3.3.1
a) `map even :: Integral a => [a] -> [Bool]`
b) `map head . snd :: (a, [[b]]) -> [b]`
c) `filter ((4 >) . last) :: (Ord a, Num a) => [[a]] -> [[a]]`
d) `const const :: a -> b -> c -> b`

Řeš. 3.3.2

```
countStudentsByPoints :: Integer -> [(String, Integer)] -> Int
countStudentsByPoints pt s = length (filter (== pt) (map snd s))

countStudentsByPoints' :: Integer -> [(String, Integer)] -> Int
countStudentsByPoints' pt = length . filter ((== pt) . snd)

studentNamesByPoints :: Integer -> [(String, Integer)] ->
 ↪ [String]
studentNamesByPoints pt s = getNames (filter ((== pt) . snd) s)

studentsStartingWith :: Char -> [(String, Integer)]
                     -> [(String, Integer)]
studentsStartingWith c = filter ((== c) . head . fst)
```

Řeš. 3.3.3 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))
```

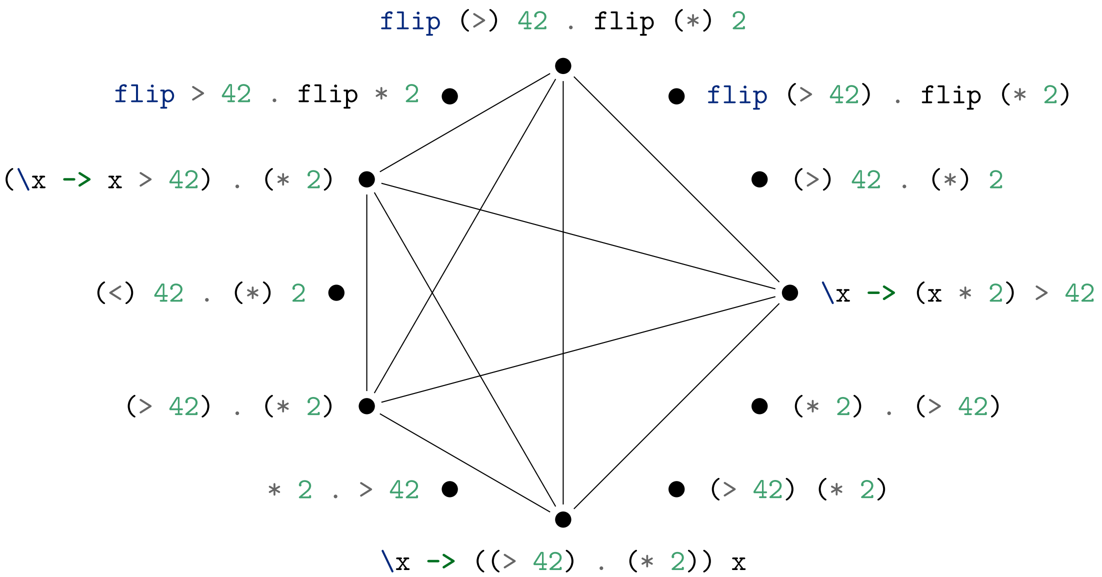
d)

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

e)

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

Řeš. 3.3.4 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. 3.3.5 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. 3.3.7 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. 3.3.8 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. 3.3.9 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. 3.3.10 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. 3.3.11 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. 3.3.12 Několikrát po sobě použijeme funkci `flip`.

```
g = flip (flip ... (flip (flip f c1) c2) ... cn)
```