Cvičení 4

4.1 Další funkce nad seznamy

Příklad 4.1.1 S pomocí interpretru zjistěte typy funkcí `and, or, all` a `any`. Zkuste je vyhodnotit na nějakých parametrech a přijít na to, co počítají (jejich název je vhodnou nápovědou).

Příklad 4.1.2 Zjistěte, co dělají následující funkce:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

Příklad 4.1.3 Funkci `zip :: [a] -> [b] -> [(a,b)]` lze definovat následovně:

```
zip (x:s) (y:t) = (x,y) : zip s t
zip _     _     = []
```

a) Které dvojice parametrů vyhovují prvnímu řádku definice?
b) Přepište definici tak, aby první klauzule definice (první řádek) byla použita jako poslední klauzule definice.

Příklad 4.1.4 Definujte funkci `zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]`.

Příklad 4.1.5 Funkce `unzip :: [(a,b)] -> ([a],[b])` může být definována následovně:

```
unzip [] = ([],[])
unzip ((x,y):s) = (x:u,y:v) where (u,v) = unzip s
```

Definujte analogicky funkce `unzip3`, `unzip4`, ...

Příklad 4.1.6 Jaká je hodnota následujících výrazů?

a) `zipWith (^) [1..5] [1..5]`
b) `zipWith (:) "MF" ["axipes", "ík"]`
c) `let fibs = [0,1,1,2,3,5,8,13] in zipWith (+) fibs (tail fibs)`
d) `let fibs = [0,1,1,2,3,5] in zipWith (/) (tail (tail fibs)) (tail fibs)`

Příklad 4.1.7 Definujte funkci `zip` pomocí funkce `zipWith`.

Příklad 4.1.8 Naprogramujte funkci `connectEven :: Integral b => [a] -> [b] -> [(a, b)]`, která dostane dva seznamy a vrátí jeden seznam dvojic `(x, y)`, kde `x` je prvek z prvního seznamu na i-té pozici, právě když `y` je sudý prvek z druhého seznamu na i-té pozici.

Pokud je jeden ze seznamů kratší, přesahující prvky z delšího seznamu ignorujte.

`connectEven [] [9, 10]` $\rightsquigarrow^*$ `[] connectEven ['b', 'c', 'd'] [2, 3]` $\rightsquigarrow^*$ `[('b', 2)]`

Příklad 4.1.9 Napište funkci, která zjistí, jestli jsou v seznamu typu `Eq a => [a]` některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

4.2 η-redukce, pointfree vs. pointwise zápis

Příklad 4.2.1 Uvažme funkci `negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních predikátů (funkcí typu `a -> Bool`). Tj. funkce `negp` vrátí opačnou logickou hodnotu, než by vrátil zadaný predikát na zadané hodnotě.

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

Příklad 4.2.2 Následující výrazy použijte v lokální definici a vyhodnoťte v interpretru jazyka Haskell na vhodných parametrech. Po úspěšné aplikaci výrazy upravujte tak, abyste se při jejich definici vyhnuli použití λ-abstrakce a formálních parametrů.

a) `\x -> 3 * x`
b) `\x -> x ^ 3`
c) `\x -> [x]`
d) `\x y -> x ^ y`
e) `\x y -> y ^ x`

Příklad 4.2.3 Převeďte následující funkce do pointfree tvaru:

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

Příklad 4.2.4 Převeďte následující výrazy do pointwise tvaru:

a) `(^2) . mod 4 . (+1)`
b) `(+) . sum . take 10`
c) `map f . flip zip [1, 2, 3]` (funkce `f` je definována externě)
d) `(.)`

Příklad 4.2.5 Určete typ následujících funkcí. Přepište tyto definice funkcí tak, abyste v jejich definici nepoužili λ-abstrakci a formální parametry (tj. chce se pointfree definice).

a) `f x y = y`
b) `h x y = q y . q x`

Příklad 4.2.6 Zjistěte, co dělají následující funkce a určete jejich typ:

a) `h1 = (.(,)) . (.) . (,)`

b) `h2 = ((,).) . (,)`

Příklad 4.2.7 Určete typ následujících výrazů. Pak je převeďte z pointfree tvaru do pointwise tvaru (vytvořte z nich λ-funkce s dostatečným množstvím parametrů) a upravte je do co možno nejčitelnější podoby. Dejte si pozor, aby byl výraz po převodu ekvivalentní původnímu výrazu (významově i typově).

a) `flip const map`
b) `flip . const map`
c) `flip const . map`
d) `flip . const . map`
e) `flip (.) const map`
f) `flip const (.) map`
g) `flip (.) const . map`
h) `flip . const (.) map`
i) `flip (.) const (.) map`

Příklad 4.2.8 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).

Příklad 4.2.9 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`

Příklad 4.2.10 Převeďte následující funkce do pointfree tvaru:

a) `\x -> f . g x`
b) `\f -> flip f x`
c) `\x -> f x 1`
d) `\x -> f 1 x True`
e) `\x -> f x 1 2`
f) `const x`

Příklad 4.2.11 Převeďte následující funkce do pointfree tvaru:

a) `\x -> 0`
b) `\x -> zip x x`
c) `\x -> if x == 1 then 2 else 0`
d) `\_ -> x`

Někde je nutné použít funkce `const` a `dist`:

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

4.3 Líné vyhodnocování a práce s nekonečnými seznamy

Příklad 4.3.1 Naprogramujte funkci `naturals`, která bude generovat seznam všech přirozených čísel.

Příklad 4.3.2 Uvažte význam líného vyhodnocování v následujících výrazech:

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

Příklad 4.3.3 Definujte funkce `cycle` a `replicate` pomocí jednodušších funkcí (lze při tom použít funkci `repeat`).

Příklad 4.3.4 Pomocí některé z funkcí `iterate`, `repeat`, `replicate`, `cycle` vyjádřete nekonečné seznamy:

a) Seznam sestávající z hodnot `True`.
b) Rostoucí seznam všech mocnin čísla 2.
c) Rostoucí seznam všech mocnin čísla 3 se sudým exponentem.
d) Rostoucí seznam všech mocnin čísla 3 s lichým exponentem.
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,...]`.

Příklad 4.3.5 Definujte Fibonacciho posloupnost, tj. seznam 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`).

Příklad 4.3.6 Pomocí rekurzivní definice a funkce `zipWith` vyjádřete Fibonacciho posloupnost.

Příklad 4.3.7 Elegantním způsobem zapište nekonečný výraz $p = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots$

4.4 Intensionální seznamy

Příklad 4.4.1 Pomocí znalostí z minulých cvičení definujte funkci `fstOdd2s`, která pro daný seznam vygeneruje seznam všech dvojic prvků z tohoto seznamu takových, kde první člen je lichý.

Příklad 4.4.2 Pomocí intensionálních seznamů definujte funkci `divisors`, která k zadanému přirozenému číslu vrátí seznam jeho kladných dělitelů.

Příklad 4.4.3 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],...]`
f) seznam všech dvojic přirozených čísel (zadefinujte tak, aby se ke každému prvku dalo dopočítat v konečném čase)

Příklad 4.4.4 Je možné zapsat intensionálním způsobem prázdný seznam? Pokud ano, jak, pokud ne, proč?

Příklad 4.4.5 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)`

Příklad 4.4.6 Intensionálním způsobem zapište následující seznamy nebo funkce:

a) rostoucí seznam druhých mocnin kladných celých čísel menších než 1000, které po dělení 7 dají zbytek 2
b) `[[1],[2,2,2],[3,3,3,3,3],[4,4,4,4,4,4,4],...]` (hledejte vztah mezi číslem a počtem jeho výskytů)
c) `["z","yy","xxx",...,"aaa...aaa"]` (znak `a` se v posledním členu vyskytuje přesně 26krát)
d) následující seznam „2D matic“

```
[ [[1]],
  [[1,1],[1,1]],
  [[1,1,1],[1,1,1],[1,1,1]], ... ]
```

Příklad 4.4.7 Napište funkci, která ze seznamu prvků vygeneruje všechny

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

Výsledný seznam ať je lexikograficky seřazen. Tam, kde je to nutné, můžete předpokládat, že prvky seznamu jsou různé. Také se můžete v případě potřeby omezit na seznamy s porovnatelnými prvky (tj. typu `Eq a => a`).

Příklad 4.4.8 Co dělají následující funkce? Nejdříve určete jejich typy.

a) `\s -> [ h:t | h <- head s, t <- tail s]`
b) `\s -> [ h:t | t <- tail s, h <- head s]`

c) `\s -> [ h:t | h <- map head s, t <- map tail s]`
d) `\s -> [ h:t | t <- tail s, let h = head t]`

Příklad 4.4.9 Přepište intensionálně zapsané seznamy pomocí funkcí `filter`, `map`, `concat`, ... a opačně.

a) `\s -> [ t | t <- replicate 2 s, even t]`
b)
```
\s -> map (\m -> (m, m^2)) $ filter isPrime $
  map (\x -> 2 * x + 1) $ filter acceptable s
```
c) `concat`
d) `map (+1) . filter even . concat . filter valid`

Příklad 4.4.10 Která z níže uvedených funkcí je časově efektivnější? Proč?

```
f1 :: [a] -> [a]
f1 s = [ s !! n | n <- [0,2..length s] ]

f2 :: [a] -> [a]
f2 (x:_:s) = x : f2 s
f2 _ = []
```

Příklad 4.4.11 Uvažujme datový typ matic zapsaných ve formě seznamu seznamů prvků matice (po řádcích):

```
type Matrix a = [[a]]
```

Implementujte tyto funkce:

a) `add` – sčítá dvě matice
b) `transpose` – transponuje zadanou matici
c) `mult` – vynásobí dvě matice

Vždy předpokládejte, že matice jsou zadány korektně (každý podseznam má stejnou délku), a že u sčítání a násobení mají matice kompatibilní rozměry.

Pokud uznáte za vhodné, můžete použít některou funkci při definici jiné.

Řešení

Řešení 4.1.1 Lze nalézt v oficiální dokumentaci: http://hackage.haskell.org/package/base/docs/Prelude.html#v:and.

Řešení 4.1.2 http://hackage.haskell.org/package/base/docs/Prelude.html#v:takeWhile

Řešení 4.1.3

a) Na základě vzorů vidíme, že u obou parametrů jde o seznam s alespoň jedním prvkem, tedy řádek se použije, pokud oba argumenty jsou neprázdné seznamy.
b) Musíme zachytit případy, kdy alespoň jeden ze seznamů je prázdný:

```
zip []     _      = []
zip _      []     = []
zip (x:s) (y:t) = (x,y) : zip s t
```

Řešení 4.1.4 Lze se snadno inspirovat definicí funkce `zip`:

```
zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3 (x:s) (y:t) (z:u) = (x,y,z) : zip3 s t u
zip3 _     _     _     = []
```

Řešení 4.1.5

```
unzip3 :: [(a,b,c)] -> ([a],[b],[c])
unzip3 []          = ([],[],[])
unzip3 ((x,y,z):s) = (x:u,y:v,z:w) where (u,v,w) = unzip3 s

unzip4 :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip4 []            = ([],[],[],[])
unzip4 ((x,y,z,q):s) = (x:u,y:v,z:w,q:t) where (u,v,w,t) = unzip4 s

...
```

Řešení 4.1.6

a) $\rightsquigarrow^*$ `[1,4,27,256,3125]`
b) $\equiv$ `zipWith (:) ['M', 'F'] ["axipes", "ík"]` $\rightsquigarrow^*$ `["Maxipes", "Fík"]`
c)

$\rightsquigarrow^*$ `zipWith (+) [0,1,1,2,3,5,8,13] [1,1,2,3,5,8,13]` $\rightsquigarrow^*$
$\rightsquigarrow^*$ `[1,2,3,5,8,13,21]`

d)

$\rightsquigarrow^*$ `zipWith (/) [1,2,3,5] [1,1,2,3,5]` $\rightsquigarrow^*$ `[1.0,2.0,1.5,1.666]`

Řešení 4.1.7 `zip = zipWith (,)`

Řešení 4.1.8 Řešením je využití funkce `zip`, kterou spojíme dané dva seznamy do jedného seznamu dvojic. Ten už jenom vyfiltrujeme podle zadané podmínky:

```
connectEven xs ys = filter (even . snd) (zip xs ys)
```

Řešení 4.1.9

```
f1 :: Eq a => [a] -> Bool
f1 (x:y:s) = x == y || f1 (y:s)
f1 _       = False
```

Nebo kratší řešení používajíce funkci `zipWith` a `or` (udělá logický součet všech hodnot v zadaném seznamu):

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

Řešení 4.2.1

a) Naším cílem je ze zadané funkce vytvořit negovanou funkci. Z typu funkce `negp` vidíme, že ji lze zapsat tak, že uvedeme jak funkční argument, tak argument s hodnotou. Pak jen výsledek volání `f` obalíme funkcí `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.
Pak 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
```

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

Řešení 4.2.2

a)

```
(\x -> 3 * x) (-4)
```

```
\x -> 3 * x
\x -> (*) 3 x
(*) 3
```

nebo

```
\x -> 3 * x
\x -> (3*) x
(3*)
```

Lze si vybrat, jestli použijeme prefixový zápis operátoru nebo operátorovou sekci.

b)

```
(\x -> x ^ 3) 5.1

\x -> x ^ 3
\x -> (^) x 3
\x -> flip (^) 3 x
flip (^) 3
```

nebo

```
\x -> x ^ 3
\x -> (^3) x
(^3)
```

Opět lze vybrat mezi prefixovým zápisem operátoru pomocí `flip` nebo operátorovou sekcí.

c)

```
(\x -> [x]) [[34]]

\x -> [x]
\x -> x:[]
\x -> (:[]) x
(:[])
```

nebo

```
\x -> [x]
\x -> x:[]
\x -> (:) x []
\x -> flip (:) [] x
flip (:) []
```

d)

```
(\x y -> x ^ y) 2 2000

\x y -> x ^ y
\x y -> (^) x y
\x -> (^) x
(^)
```

e)

```
(\x y -> y ^ x) 2 2000

\x y -> y ^ x
\x y -> (^) y x
\x y -> flip (^) x y
\x -> flip (^) x
```

```
flip (^)
```

Tady bychom mohli postupovat následovně v domnění, že se vyhneme použití `flip`:

```
\x y -> y ^ x
\x y -> (^x) y
\x -> (^x)
```

Avšak teď narazíme na problém, že operátorovou sekci nelze upravit tak, abychom mohli provést η-redukci. Východiskem z této situace je přepsat `(^x)` na `flip (^) x`, čímž se však dostáváme k prvně uvedenému řešení. Problém s použitím operátorové sekce nenastane, pokud její operand nebude obsahovat formální argument, který budeme potřebovat odstranit – tedy lze je bez obav použít třeba u číselných operandů.

Řešení 4.2.3

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
```

Řešení 4.2.4

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)
```

Řešení 4.2.5

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) V tomto případě si třeba dát pozor na funkci `q`. Vyskytuje se tady ve dvou různých výskytech a ty mohou (a také i budou) mít odlišné typy (situace podobná jak u `head . head`.

Také pozor na to, že pokud by jste chtěli určit typ v interpretu a upravili by jste si funkci na `\q x y -> q y . q x`, nedostanete správný výsledek. To je dáno tím, že zadáním funkce `q` jako argumentu vynutíte stejný typ pro všechny její výskyty.

`h :: a -> b -> c -> d`

Převod na pointfree tvar:

```
h x y = q y . q x
h x y = (q y) . (q x)
h x y = (.) (q y) (q x)
h x y = flip (.) (q x) (q y)
h x y = (flip (.) (q x)) (q y)
h x y = (flip (.) (q x) . q) y
h x = flip (.) (q x) . q
h x = (flip (.) (q x)) . q
h x = (.q) (flip (.) (q x))
h x = ((.q) . flip (.) . q) x
h = (.q) . flip (.) . q
```

Řešení 4.2.6

a) Nejsnáze to zjistíme převodem na pointwise tvar, který je přehlednější:

```
(.(,)) . (.) . (,)
```

Máme složení funkcí, které vyžaduje argument, takže ho dodáme.

```
\x -> ((.(,)) . (.) . (,)) x
```

Rozepíšeme výraz dle definice funkce na nejvyšší úrovni, tedy tečky. Tady lze tuto úpravu zkrátit a rozepsat obě tečky naráz, tedy `(f . g . h) x` ≡ `f (g (h x))`.

```
\x -> (.(,)) ((.) ((,) x))
```

Opět rozepíšeme funkci na nejvyšší úrovni, tedy `(.(,))` na začátku výrazu.

```
\x -> ((.) ((,) x)) . (,)
```

Tečka zas vyžaduje argument – dodáme.

```
\x y -> (((.) ((,) x)) . (,)) y
```

A rozepíšeme dle definice tečky.

```
\x y -> ((.) ((,) x)) ((,) y)
```

Odstraníme implicitní závorky.

```
\x y -> (.) ((,) x) ((,) y)
```

Přepis tečky do infixu.

```
\x y -> (,) x . (,) y
```

Opět tečka a přidání argumentu.

```
\x y z -> ((,) x . (,) y) z
```

Rozepsání dle definice tečky.

```
\x y z -> (,) x ((,) y z)
```

Přepis do „infixového“ tvaru operátoru na tvorbu uspořádaných dvojic.

```
\x y z -> (x, (y, z))
```

Tedy odsud vidíme celkem jasně, co funkce `h1` dělá, a také snadno určíme její typ:

```
h1 :: a -> b -> c -> (a, (b, c))
```

Alternativně k tomuto postupu je zde možnost označit si v původním zadání skládané funkce jako `f, g, h`, což zvýší přehlednost. Pak lze upravovat tento výraz a vždy když narazíme na potřebu použít některou z těchto funkcí, rozepíšeme ji do původního tvaru.

b) Opět postupujeme podobně:

```
((,).) . (,)
```

Tečka vyžaduje argument.

```
\x -> (((,).) . (,)) x
```

Odstranění implicitních závorek.

```
\x -> ((,).) ((,) x)
```

Rozepsání výrazu na nejvyšší úrovni – operátorové sekce.

```
\x -> (,) . ((,) x)
```

Tečka vyžaduje argument – dodáme.

```
\x y -> ((,) . ((,) x)) y
```

Rozepíšeme vnitro závorek dle definice tečky.

```
\x y -> (,) (((,) x) y)
```

Odstranění implicitních závorek a přepis do přirozeného infixového tvaru.

```
\x y -> (,) (x, y)
```

Dodání požadovaného argumentu.

```
\x y z -> (,) (x, y) z
```

Přepis do přirozeného infixového tvaru.

```
\x y z -> ((x, y), z)
```

Typ je `h2 :: a -> b -> c -> ((a, b), c)`.

Řešení 4.2.7 Typy zadaných výrazů jsou následující:

a) `a -> a`
b) `a -> [b] -> (b -> c) -> [c]`
c) `(a -> b) -> c -> c`
d) `(a -> b) -> [a] -> c -> [b]`
e) `a -> [b] -> [a]`
f) `(a -> b) -> [a] -> [b]`
g) výraz není typově správně utvořen
h) `(a -> c) -> b -> (b -> a) -> c`
i) `(a -> b) -> a -> (c -> d) -> [c] -> [d]`

Po převodu do pointwise tvaru a přepsání funkcí dle definic dostaneme funkce uvedené níže. Navzdory tomu, že některé vypadají, že by se ještě dali zjednodušit, například podpříklad c), toto zjednodušení není možné vykonat, protože by nezachovalo typovou ekvivalenci.

a)

```
flip const map
\x -> flip const map x
\x -> const x map
\x -> x
```

b)

```
flip . const map
\x -> (flip . const map) x
\x -> flip (const map x)
\x y z -> flip (const map x) y z
\x y z -> (const map x) z y
\x y z -> const map x z y
\x y z -> map z y
```

c)

```
flip const . map
\x -> (flip const . map) x
\x -> flip const (map x)
\x y -> flip const (map x) y
\x y -> const y (map x)
\x y -> const y (\z -> map x z)
```

Funkci `const` nelze dále rozepsat dle definice, protože bychom přišli o informaci, že `x` musí být funkčního typu.

d)

```
flip . const . map
\x -> (flip . const . map) x
\x -> flip (const (map x))
\x y z -> flip (const (map x)) y z
\x y z -> (const (map x)) z y
\x y z -> const (map x) z y
```

```
\x y z -> (map x) y
\x y z -> map x y
```

e)

```
flip (.) const map
(.) map const
map . const
\x -> (map . const) x
\x -> map (const x)
\x y -> map (const x) y
\x y -> map (\z -> const x z) y
```

f)

```
flip const (.) map
const map (.)
map
\x y -> map x y
```

g)

```
flip (.) const . map
\x -> (flip (.) const . map) x
\x -> flip (.) const (map x)
\x -> (.) (map x) const
\x -> map x . const
\x y -> (map x . const) y
\x y -> map x (const y)
```

Výraz `map x` očekává jako argument seznam, avšak `const y` je funkce. Výraz tedy není typově správně utvořen.

h)

```
flip . const (.) map
\x -> (flip . const (.) map) x
\x -> flip (const (.) map x)
\x y z -> flip (const (.) map x) y z
\x y z -> (const (.) map x) z y
\x y z -> const (.) map x z y
\x y z -> (.) x z y
\x y z -> x (z y)
```

i)

```
flip (.) const (.) map
(flip (.) const (.)) map
((.) (.) const) map
(.) (.) const map
(.) (const map)
\x y -> (.) (const map) x y
\x y -> const map (x y)
\x y -> const (\z q -> map z q) (x y)
```

Opět nelze rozepsat `const`, jelikož bychom ztratili informaci o tom, že `x` je funkce a její první argument má stejný typ jako `y`.

Řešení 4.2.8 Několikrát po sobě použijeme funkci `flip`.
`g = flip (flip ... (flip (flip f c1) c2) ... cn)`

Řešení 4.2.9 Nejdříve vhodným použitím funkcí `const` a `flip` zabezpečíme, aby se nám všechny tři formální argumenty nacházely i na pravé straně všech tří funkcí:

```
f1 x y z = const (const x y) z
f2 x y z = flip const x (const y z)
f3 x y z = flip const x (flip const y z)
```

Pak už jenom mechanicky převedeme získané výrazy do pointfree tvaru.

```
f1 = (.) const . const
f2 = flip (.) const . (.) . flip const
f3 = flip (.) (flip const) . (.) . flip const
```

Řešení 4.2.10

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

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

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

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

e)
```
\x -> f x 1 2
\x -> (f x 1) 2
\x -> (flip f 1 x) 2
\x -> (flip f 1) x 2
\x -> flip (flip f 1) 2 x
flip (flip f 1) 2
```

f)
```
const x
```
Parametr `x` samozřejmě nelze odstranit, protože není vázán λ-abstrakcí.

Řešení 4.2.11

a)
```
\x -> 0
\x -> const 0 x
const 0
```
b)
```
\x -> zip x x
\x -> zip x (id x)
\x -> dist zip id x
dist zip id
```
c) Není možno převést, poněvadž `if-then-else` není klasická funkce, ale syntaktická konstrukce, podobně jako `let-in`.

d)
```
\_ -> x
\t -> x
\t -> const x t
const x
```

Řešení 4.3.1 Díky línému vyhodnocování v Haskellu lze generovat nekonečné datové struktury a dokonce s nimi pracovat.

Nemusíme proto ani mít k dispozici speciální syntaxi, protože se dají napsat *indukční definicí*, resp. neohraničenou rekurzivní funkcí, co Haskellu nedělá problémy:

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

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

Řešení 4.3.2

a) Díky lenosti funkce `take` se nemusíme podívat na více než prvních deset prvků výrazu `[1..]`, přičemž každý z nich lze získat v konečném čase. Tedy navzdory nekonečnosti seznamu `[1..]` dojde k vyhodnocení zadaného výrazu v konečném čase.
b) 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.
c) Funkce `f` je definována jenom pro prázdný seznam, ale ve výrazu je volána na neprázdném seznamu. Normálně bychom dostali chybovou zprávu `Non-exhaustive patterns in function f`, ale díky lenosti vyhodnocování `const` nedojde k tomuto volání a vyhodnocení skončí bez chyby.
d) 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. Navíc výraz tohoto typu by v matematice stejně neměl definovanou hodnotu.

e) 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 případný problém tohoto typu je vždy zaznamenán a líné vyhodnocování situaci nezachrání.

Řešení 4.3.3

```
cycle = concat . repeat
replicate n = take n . repeat
```

Řešení 4.3.4

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]
```

Řešení 4.3.5 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.

Řešení 4.3.6

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

Řešení 4.3.7 `p = sum (zipWith (/) (iterate negate 4) [1,3..])`

Řešení 4.4.1 Nejpřímočarejší řešení je vytvořit si pomocní rekurzivní funkci, která si bude udržovat původní seznam.

To umožní, abychom mohli mapovat každý prvek na všechny ostatní (včetně sama sebe). Potom to už je jenom o tom, na co vlastně chceme mapovat, jak výsledky spojit a vyfiltrovat ty správné:

```
allTuples :: Integral a => [a] -> [a] -> [(a, a)]
allTuples [] _ = []
allTuples (x:xs) r = map ((,) x) r ++ allTuples xs r

fstOdd2s :: Integral a => [a] -> [(a, a)]
fstOdd2s xs = filter (odd . fst) (allTuples xs xs)
```

Jedno z elegantnějších řešení je využít funkce `concatMap`, která spojování dělá za nás. Namapuje každý prvek původního seznamu na právě jeden seznam a následně tyto seznamy spojí dohromady (vytvoří jeden nový seznam s prvky z každého z vytvořených seznamů).

Pro naše účely je to užitečné, protože tímto způsobem můžeme každý prvek *namapovat na mapu dvojic se všemi ostatními včetně sebe*, kde vystupuje jako první. A to je to stejné, což jsme již dělali:

```
fstOdd2s :: Integral a => [a] -> [(a, a)]
fstOdd2s xs = filter (odd . fst) (concatMap (\x -> map ((,) x) xs) xs)
```

Nakonec dodáváme, že ani jedno z těchto řešení není efektivní, protože dochází ke spojování seznamů `(++)`. Pro efektivnější `allTuples` můžeme použít například intensionální seznamy nebo foldy (později).

Řešení 4.4.2

```
divisors :: Int -> [Int]
divisors n = [ x | x <- [1..n], mod n x == 0 ]
```

Řešení 4.4.3

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..] ]`
f) `[ (y,x-y) | x <- [2..], y <- [1..(x-1)] ]`

Řešení 4.4.4 A proč ne. :) Minimálně příklad s intensionálním zápisem seznamových funkcí dává několik inspirací. Uveďme některá možná řešení:

```
[ 0 | _ <- [] ]                    -- pozor typ Num a => [a]
[ 0 | False ]                      -- opet typ  Num a => [a]
[ undefined | _ <- [] ]            -- typ OK: [a]
[ undefined | _ <- [1..10], False ] -- typ OK
```

Poznámka: fukce `undefined :: a` je polymorfní konstanta, jejíž vyhodnocení vždy způsobí chybu (obdobně jako u funkce `error`).

Také poznamenejme, že tato řešení jsou nesprávná:

a) `[ | ]` – syntaktická chyba, před i za svislítkem musí být výraz
b) `[ | x <- s ]` – obdobně jako první příklad a taky `s` není definováno
c) `[ x | ]` – obdobně jako první příklad, navíc `x` není definováno

Řešení 4.4.5

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) ]`

Řešení 4.4.6

a) `[ x^2 | x <- [1..999], mod x 7 == 2 ]`
b) `[ [n | _ <- [1..(2*n-1)]] | n <- [1..] ]`
c) `[ [c | _ <- [1..k] ] | (k, c) <- zip [1..26] ['z','y'..'a'] ]`
d) `[ [ [1|_<-[1..n]] | _ <- [1..n]] | n <- [1..] ]`

Řešení 4.4.7

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.

Řešení 4.4.8 Všechny funkce jsou typu `[[a]] -> [[a]]`. Pro jednoduchost označme `s = x:xs`.

a) Funkce spáruje všemi způsoby prvky `x` s prvky `xs`.
b) Podobně jako a), jenom nejdříve zpracovává prvky `xs`. Výsledek tedy dostaneme v jiném pořadí.
c) Funkce spáruje všemi způsoby „hlavy“ prvků `s` s „chvosty“ prvků `s`.
d) Funkce vrátí seznamy z `xs` (kromě prvního) s duplikovanými prvními prvky.

Řešení 4.4.9

a) `filter even . replicate 2`
b) `\s -> [(t, t^2) | x <- s, acceptable x, let t = 2 * x + 1, isPrime t]`
c) `\s -> [t | x <- s, t <- x]`
d) `\s -> [t + 1 | x <- s, valid x, t <- x, even t]`

Řešení 4.4.10 Nechť $n =$ `length s`. Lepší časovou složitost má funkce `f2`, protože projde seznamem jenom jednou, tedy celkově v čase $\mathcal{O}(n)$. Na druhé straně `f1` vykoná nejvíce $n/2+1$ volání funkce `(!!)`. Tyto volání se v tomhle případě vykonají každé v čase $\mathcal{O}(k)$, kde k je druhý argument funkce `(!!)`. Dohromady tedy vyžaduje čas $\mathcal{O}(n^2)$.

Řešení 4.4.11

a)

```
add :: Num a => Matrix a -> Matrix a -> Matrix a
add = zipWith (zipWith (+))
```

b)

```
transpose :: Matrix a -> Matrix a
transpose = foldr (zipWith (:)) (repeat [])
```

c)

```
mult :: Num a => Matrix a -> Matrix a -> Matrix a
mult m1 m2 = [[sum (zipWith (*) x y) | y <- transpose m2] | x <- m1]
```