Fakulta informatiky
Masarykovy univerzity

Sbírka řešených úloh

IB015 Neimperativní programování

Poslední modifikace: **24. ledna 2015**

Chyby, překlepy, nejasnosti a nejednoznačnosti prosím nahlašujte v diskusním fóru předmětu.

Tento text vznikl přepsáním a restrukturalizací mnoha materiálů, sbírek a přednášek o funkcionálním a logickém programování. Autoři příkladů a jejich řešení jsou Jiří Barnat, Marek Klučár, Tomáš Szaniszlo, Libor Škarvada, Vladimír Štill, Martin Ukrop a nesčetní další vyučující bývalých předmětů *IB015 Úvod do funkcionálního programování* a *IA013 Logické programování*.

> *A Cup of Tea*
>
> *Nan-in, a Japanese master during the Meiji era (1868-1912), received a university professor who came to inquire about Zen. Nan-in served tea. He poured his visitor's cup full, and then kept on pouring.*
>
> *The professor watched the overflow until he no longer could restrain himself. ``It is overfull. No more will go in!''*
>
> *``Like this cup,'' Nan-in said, ``you are full of your own opinions and speculations. How can I show you Zen unless you first empty your cup?''*

The hardest thing about learning functional programming is forgetting what you think you ``know''. It might be true, but not in this context. Everything is different, so you need to start from the very beginning.

If you know C++, and you want to learn Java, you compare both languages all the time, and this works fine. It's the same with natural languages: If you know English and want to learn Spanish, it's okay to compare both languages all the time, as they follow the same indo-germanic paradigm. But if you want to learn something fundamental different, like Japanese, or Haskell, you have to ``unlearn'' first. You won't make real progress until you stop comparing.

Zenový kōan a komentář o výuce funkcionálního programování přebrán z diskusního fóra *Programmers Stack Exchange* od uživatele *Landei*.

Obsah

Cvičení 1 4
1.1 Priority operátorů, prefixový a infixový zápis . . . 4
1.2 Definice funkcí podle vzoru . . . 5
1.3 Lokální definice a globální definice . . . 6

Cvičení 2 7
2.1 Datové typy . . . 7
2.2 Částečná aplikace . . . 8
2.3 Funkce na seznamech . . . 8

Cvičení 3 11
3.1 Skládání funkcí . . . 11
3.2 Typování funkčních aplikací a definic . . . 11
3.3 Další funkce na seznamech . . . 13

Cvičení 4 15
4.1 λ-abstrakce . . . 15
4.2 η-redukce, pointfree vs. pointwise zápis . . . 16
4.3 Curryfikace . . . 17

Cvičení 5 18
5.1 Akumulační funkce na seznamech . . . 18
5.2 Líné vyhodnocování a práce s nekonečnými seznamy . . . 21

Cvičení 6 23
6.1 Úvod do IO . . . 23
6.2 IO pomocí `do`-notace . . . 23
6.3 IO pomocí operátorů `>>=` a `>>` . . . 24
6.4 Vlastní datové typy . . . 25

Cvičení 7 28
7.1 Typové třídy . . . 28
7.2 `Maybe` . . . 28
7.3 Rekurzivní datové typy . . . 29

Cvičení 8 32
8.1 Intensionální seznamy . . . 32
8.2 Katamorfismy nad vlastními datovými typy . . . 33

Cvičení 9 38
9.1 Opakování . . . 38
9.2 Složitější příklady . . . 42

Cvičení 10 45
10.1 Úvod do Prologu, backtracking . . . 45

10.2 Unifikace . . . 45
10.3 Backtracking a SLD stromy . . . 47

Cvičení 11 **49**
11.1 Číselné operace . . . 49
11.2 Práce se seznamy . . . 51

Cvičení 12 **53**
12.1 Řezy . . . 53
12.2 Negace . . . 54
12.3 Predikáty pro všechna řešení . . . 54
12.4 Základy I/O . . . 55

Cvičení 13 **56**
13.1 Logické programování s omezujícími podmínkami . . . 56
13.2 Opakování . . . 57

A Přílohy **145**
A.1 pt.hs . . . 145
A.2 guess.hs . . . 145
A.3 BinTree.hs . . . 146
A.4 treeFold.hs . . . 146
A.5 pedigree.pl . . . 147
A.6 einstein.pl . . . 148
A.7 einsteinSol.pl . . . 152
A.8 einsteinSol2.pl . . . 153
A.9 sudoku.pl . . . 155

Cvičení 1

1.1 Priority operátorů, prefixový a infixový zápis

Příklad 1.1.1 S použitím interpretru 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) `(3 == 3) == 3` versus `(4 == 4) == (4 == 4)`

Příklad 1.1.2 S využitím interního příkazu `:info` interpretru `ghci` zjistěte prioritu a směr vyhodnocování následujících operací:

``., !!, ^, *, /, `div`, `mod`, +, -, :, ++, ==, /=, >, <, >=, <=, &&, ||``

Příklad 1.1.3 Vysvětlete, co je chybné na následujících podmíněných výrazech, a výrazy vhodným způsobem upravte.

a) `if 5 - 4 then False else True`
b) `if 0 < 3 && odd 6 then 1 else "chyba"`
c) `(if even 8 then (&&)) (0 > 7) True`

Příklad 1.1.4 Přepište infixové zápisy výrazů do syntakticky správných prefixově zapsaných výrazů a naopak:

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

Příklad 1.1.5 Doplňte všechny implicitní závorky do následujících výrazů:

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

Příklad 1.1.6 Zjistěte (bez použití interpretru), na co se vyhodnotí následující výraz. Poté jej přepište do prefixového tvaru a pomocí interpretru ověřte, že se jeho hodnota nezměnila.
``5 + 7 * 5 `mod` 3 `div` 2 == 3 * 2 - 1``

Příklad 1.1.7 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``

Příklad 1.1.8 Které z následujících výrazů jsou korektní?

a) `((+) 3)`

b) `(3 (+))`
c) `(3+)`
d) `(+3)`
e) `(.(+))`
f) `(+(.))`
g) `(.+)`
h) `(+.)`
i) `.even`
j) `(.((.).))`

1.2 Definice funkcí podle vzoru

Příklad 1.2.1 Definujte funkci `logicalAnd`, která se chová stejně jako funkce logické konjunkce, tak, abyste v definici

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

Příklad 1.2.2 Definujte rekurzivní funkci pro výpočet faktoriálu.

Příklad 1.2.3 Upravte následující kód tak, aby funkce pro záporná čísla necyklila, ale skončila s chybovou hláškou. Použijte k tomu funkci `error :: String -> a`.

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

Příklad 1.2.4 Definujte v Haskellu funkci `dfct` (v kombinatorice někdy značenou !!), kde

$$\begin{aligned} 0!! &= 1, \\ (2n)!! &= 2 \cdot 4 \cdots (2n), \\ (2n+1)!! &= 1 \cdot 3 \cdots (2n+1) \end{aligned}$$

Příklad 1.2.5 Definujte funkci `linear` tak, že `linear a b` se vyhodnotí na řešení lineární rovnice $ax + b = 0, x \in \mathbb{R}$. Jestliže rovnice nemá právě 1 řešení, tuto skutečnost vypište na výstup pomocí funkce `error`. Před samotnou definicí funkce určete, jaký bude mít typ.

Příklad 1.2.6 Napište funkci `combinatorial` tak, že `combinatorial n k` se vyhodnotí na kombinační číslo $\binom{n}{k}$.

Příklad 1.2.7 Napište funkci `roots`, která se po aplikaci na koeficienty a, b, c vyhodnotí na počet reálných kořenů kvadratické rovnice $ax^2 + bx + c = 0$.

Příklad 1.2.8 Definujte funkci `digits`, která po aplikaci na kladné celé číslo vrátí jeho ciferný součet.

Příklad 1.2.9 Napište funkci `mygcd`, která po aplikaci na dvě kladná celá čísla vrátí jejich největšího společného dělitele. Pokuste se o co nejefektivnější implementaci.

Příklad 1.2.10 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.

Příklad 1.2.11 Napište funkci, která o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky.

Příklad 1.2.12 Co počítá následující funkce? Jak se chová na argumentech, kterými jsou nezáporná čísla? Jak se chová na záporných argumentech?

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

1.3 Lokální definice a globální definice

Příklad 1.3.1 Výraz `((3+4)/2) * ((3+4)/2 - 3) * ((3+4)/2 - 4)`

a) upravte s využitím syntaktické konstrukce pro lokální definici (`let ... in`) tak, aby se v něm neopakovaly stejné složené podvýrazy;
b) upravte stejně, ovšem s využitím globálních definic (uložených v externím souboru).

Příklad 1.3.2 Napište funkci `flipNum :: Integer -> Integer`, která vrátí číslo s ciframi v opačném pořadí. Uvažujte pouze dekadickou soustavu. Případné pomocné funkce definujte pouze lokálně.

Cvičení 2

2.1 Datové typy

Příklad 2.1.1 S pomocí interpretru určete typy následujících výrazů a najděte další výrazy stejného typu.

a) `'a'`
b) `"ahoj"`
c) `not`
d) `(&&)`
e) `(||)`
f) `True`

Příklad 2.1.2 Nalezněte příklady hodnot následujících typů:

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

Příklad 2.1.3 Určete typy následujících výrazů, zkontrolujte si řešení pomocí interpretru.

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

```
g :: Double -> Double
g x = x * x + 2
```

Příklad 2.1.4 Odstraňte všechny nadbytečné (implicitní) závorky z následujících typů:

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

Příklad 2.1.5 Co musí platit pro typ funkce `f`, aby měla funkce `g` korektní typ?

```
g x = x (f x)
```

Příklad 2.1.6 Je možné unifikovat typ zadané funkce s daným typem?

a) `id`, `a -> b -> a`
b) `const`, `(a -> b) -> a`
c) `const`, `(a -> b) -> c`
d) `map`, `a -> b -> c -> d -> e`

2.2 Částečná aplikace

Příklad 2.2.1 Co vyjadřuje výraz `min 6`? Napište ekvivalentní výraz pomocí `if`.

Příklad 2.2.2 Které z následujících výrazů jsou ekvivalentní?

a) `f 1 g 2` ≡ `f 1 (g 2)`
b) `(f 1 g) 2` ≡ `(f 1) g 2`
c) `(+ 2) 3` ≡ `2 + 3`
d) `(+) 2 3` ≡ `(+ 2) 3`
e) `81 * f 2` ≡ `(*) 81 f 2`
f) `fact n` ≡ `n * fact n - 1` (uvažujíce klasickou rekurzivní definici funkce `fact`)
g) `sin (1.43)` ≡ `sin 1.43`
h) `sin 1.43` ≡ `sin 1 . 43`
i) `8 - 7 * 4` ≡ `(-) 8 (* 7 4)`

2.3 Funkce na seznamech

Příklad 2.3.1 Rozhodněte, které z následujících seznamů jsou správně utvořené. U nesprávných rozhodněte proč, u správně utvořených určete typ. Konzultujte své řešení s interpretrem.

a) `[1, 2, 3]`
b) `(1:2):3:[]`
c) `1:2:3:[]`
d) `1:(2:(3:[]))`
e) `[1, 'a', 2]`
f) `[ [], [1, 2], 1:[] ]`
g) `[ 1, [1, 2], 1:[] ]`
h) `[]:[]`

Příklad 2.3.2 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) `[[], [""]]`

Příklad 2.3.3 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]]`

Příklad 2.3.4 Definujte funkce `myHead :: [a] -> a` (která vrátí první prvek seznamu) a `myTail :: [a] -> [a]` (která vrátí seznam bez prvního prvku). Nepoužívejte knihovní funkce `head, tail`.

Příklad 2.3.5 Definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu. Nesmíte použít funkci `last`.

Příklad 2.3.6 Definujte funkci `stripLast :: [a] -> [a]`, která pro neprázdný seznam vrátí tentýž seznam bez posledního prvku. Nesmíte použít funkci `init`.

Příklad 2.3.7 Pomocí funkce `init` definujte funkci `median`, která vrátí medián konečného uspořádaného neprázdného seznamu. Medián seznamu je jeho v pořadí prostřední prvek. Pro seznam se sudým počtem prvků vraťte levý z dvojice ve středu.

Příklad 2.3.8 Definujte funkci `len :: [a] -> Integer`, která spočítá délku seznamu. Nesmíte použít funkci `length`.

Příklad 2.3.9 Napište funkci `doubles`, která bere ze seznamu po dvou prvcích a vytváří seznam uspořádaných dvojic. Pokud má seznam lichý počet prvků, poslední prvek se zahodí.

```
doubles [1,2,3,4,5] = [(1,2), (3,4)]
doubles [0,1,2,3] = [(0,1), (2,3)]
```

Příklad 2.3.10 Definujte rekurzivní funkci `add1 :: [Integer] -> [Integer]`, která vrátí seznam, v němž je každý prvek o 1 větší než ve vstupním seznamu.

Příklad 2.3.11 Definujte rekurzivní funkci `multiplyN :: Integer -> [Integer] -> [Integer]`, která vrátí seznam, v němž je každý prvek v druhém seznamovém parametru vynásoben číslem, které je prvním parametrem funkce.

Příklad 2.3.12 Definujte rekurzivní funkci `applyToList :: (a -> b) -> [a] -> [b]`, která vezme funkci a seznam, a aplikuje danou funkci na každý prvek seznamu.

Příklad 2.3.13 Definujte funkce `add1` a `multiplyN` znovu a co nejkratším zápisem pomocí funkce `applyToList`.

Příklad 2.3.14 Definujte funkci `evens :: [Integer] -> [Integer]`, která ze seznamu čísel vybere ta sudá (párná).

Příklad 2.3.15 Definujte funkci `evens :: [Integer] -> [Integer]`, která ze seznamu vybere sudá čísla. Použijte funkci `filter`.

Příklad 2.3.16 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 interpretru) definujte novou funkci `toUpperStr`, která převádí řetězec písmen na řetězec velkých písmen, tj. `toUpperStr "bob"` $\rightsquigarrow^*$ `"BOB"`.

Příklad 2.3.17 Definujte funkci `multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2. Nepoužívejte rekurzi explicitně.

Příklad: `multiplyEven [2,3,4]` $\rightsquigarrow^*$ `[4,8]`, `multiplyEven [6,6,3]` $\rightsquigarrow^*$ `[12,12]`.

Příklad 2.3.18 Definujte funkci `sqroots :: [Double] -> [Double]`, která ze zadaného seznamu vybere kladná čísla a ta odmocní. (Využijte mimo jiné funkce `(>0)` a `sqrt`.)

Příklad 2.3.19 Vymyslete (vzpomeňte si) na další funkce pracující se seznamy, pojmenujte je v Haskellu nerezervovaným slovem, definujte je a vyzkoušejte svoji definici v interpretru jazyka Haskell. Můžete se inspirovat například funkcemi zde

http://www.postgresql.org/docs/current/static/functions-string.html.

Příklad 2.3.20 Napište funkci `fromend`, která dostane přirozené číslo x a seznam, a vrátí x-tý prvek seznamu od konce. Například `fromend 3 [1,2,3,4]` se vyhodnotí na 2. Jestli má seznam méně prvků jako x, funkce skončí s chybovou hláškou.

Příklad 2.3.21 Definujte funkci `maxima`, která dostane seznam seznamů čísel a vrátí seznam maximálních prvků jednotlivých seznamů. Například `maxima [[5,3],[2,7,13]]` se vyhodnotí na `[5,13]`. Pomozte si například funkcí `maximum`, která vrátí největší prvek seznamu.

Příklad 2.3.22 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 ["ABC","DEF"]` se vyhodnotí na `["A","E"]`.

Příklad 2.3.23 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. Například `palindrome "abbcb"` se vyhodnotí na `False` a `palindromize "abbcb"` se vyhodnotí na `"abbcbba"`.

Příklad 2.3.24 Napište funkci `brackets`, která vezme řetězec složený ze znaků `'('` a `')'` a rozhodne, jestli se jedná o korektní uzávorkování.

Příklad 2.3.25 Napište funkci `domino :: (Eq a) => [(a,a)] -> [(a,a)]`, která nějakým způsobem vybere prvky ze zadaného seznamu tak, aby měli dvojice ve výsledném seznamu vedlejší prvky stejné. Není nutné vybrat nejdelší takový podseznam. Příklad:

`domino [(1,2), (4,3), (2,5), (5,1), (2,5)]` $\rightsquigarrow$ `[(1,2), (2,5), (5,1)]`

Příklad 2.3.26 Popište, jak se chová následující funkce a pokuste se ji definovat kratším zápisem a efektivněji.

```
s2m :: Integer -> [Integer]
s2m 0 = [0]
s2m n = s2m (n - 1) ++ [ last (s2m(n - 1)) + 2 * n - 1 ]
```

Bonus: Dokažte, že je vaše nová definice ekvivalentní.

Cvičení 3

3.1 Skládání funkcí

Příklad 3.1.1 Vyhodnoťte následující výrazy:

a) `((== 42) . (2 +)) 40`
b) `((> 2) . (* 3) . ((-) 4)) 5`
c) `filter ((>= 2) . fst) [(1,"a"), (2,"b"), (3,"c")]`

Příklad 3.1.2 Uvažme funkci `negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních funkcí typu `a -> Bool` (tzv. predikátů).

a) Definujte funkci `negp`.
b) Definujte funkci `negp` jako unární funkci (s použitím pouze jednoho formálního parametru).
c) Definujte funkci `negp` bez použití formálních parametrů.

Příklad 3.1.3 Určete všechny implicitní závorky v následujících výrazech:

a) `f.g x`
b) `f (.) g (h x) . (.) f g x`

3.2 Typování funkčních aplikací a definic

Příklad 3.2.1 Určete typy výrazů:

a) `(&&) True`
b) `id "foo"`
c) `(&& False)`
d) `const True`
e) `const True False`
f) `(:[])`
g) `(:[]) True`
h) `[]:[]:[]`
i) `([]:[]):[]`

Příklad 3.2.2 Určete typy následujících výrazů:

a) `map fst`
b) `map (filter not)`
c) `const id '!' True`
d) `fst (fst, snd) (snd, fst) (True, False)`
e) `head [head] [tail] [[]]`

Příklad 3.2.3 Určete typy funkcí:

a) `swap (x,y) = (y,x)`
b) `cadr = head . tail`
c) `caar = head . head`
d) `twice f = f . f`
e) `comp12 g h x y = g (h x y)`

Příklad 3.2.4 Určete typy následujících funkcí:

a)
```
sayLength [] = "empty"
sayLength x  = "noempty"
```
b)
```
mswap True  (x, y) = (y, x)
mswap False (x, y) = (x, y)
```
c)
```
gfst (x, _) = x
gfst (x, _, _) = x
gfst (x, _, _, _) = x
```
d)
```
foo True []    = True
foo True (_:_) = False
foo False      = False
```

Příklad 3.2.5 Určete typy následujících výrazů:

a) `(+ 3)`
b) `(+ 3.0)`
c) `filter (>= 2)`
d) ``(> 2) . (`div` 3)``

Příklad 3.2.6 Určete typy následujících výrazů:

a) `id const`
b) `takeWhile (even . fst)`
c) `fst . snd`
d) `fst . snd . fst . snd . fst . snd`
e) `map . snd`
f) `head . head . snd`
g) `map (filter fst)`
h) `zipWith map`

Příklad 3.2.7 Definujte funkce tak, aby jejich nejobecnější typ byl shodný s typem uvedeným níže.

a) `f1 :: a -> (a -> b) -> (a, b)`
b) `f2 :: [a] -> (a -> b) -> (a, b)`
c) `f3 :: (a -> b) -> (a -> b) -> a -> b`
d) `f4 :: [a] -> [a -> b] -> [b]`
e) `f5 :: ((a -> b) -> b) -> (a -> b) -> b`
f) `f6 :: (a -> b) -> ((a -> b) -> a) -> b`

Příklad 3.2.8 Proč jsou první dva výrazy v pořádku (interpret je akceptuje), třetí však nikoli?

- `id id`

- `let f x = x in f f`
- `let f x = x x in f id`

Příklad 3.2.9 Určete typ funkcí `f1` až `f6` v následujících výrazech. Jestli se funkce vyskytuje ve vícero výrazech/výskytech, určete její typ jednak pro každý výraz/výskyt samostatně, a také pak unifikujte vzniklé typy (tj. zohledněte omezení na typ ze všech výrazů/výskytů).

a)
```
f1 []
snd (f1 [id])
```

b)
```
fun t = f2 ((x:y):(z:q), t)
flip (curry f2)
```

c)
```
fun s = f3 (fst f3 s + 10)
```

d)
```
(,) 1 x : f4
head f4 `elem` ['a'..'z']
```

e)
```
f5 []
1 + f5 [x:xs]
```

f)
```
f6 4
id flip f6 id
```

3.3 Další funkce na seznamech

Příklad 3.3.1 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 3.3.2 Zjistěte, co dělají následující funkce:

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

Příklad 3.3.3 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 3.3.4 Definujte funkci `zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]`.

Příklad 3.3.5 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 3.3.6 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 3.3.7 Definujte funkci `zip` pomocí funkce `zipWith`.

Příklad 3.3.8 Napište funkci, která zjistí, jestli jsou v seznamu typu `(Eq a) => [a]` některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

Cvičení 4

4.1 λ-abstrakce

Příklad 4.1.1 Které z následujících výrazů jsou korektní?

a) `\x y -> 0`
b) `\f -> f 0`
c) `(\s -> "ahoj, " ++ s, "to: " ++ s)`
d) `\x -> x . \y -> y x`
e) `\[(x, y)] z -> testIt x y z`
f) `\(_) [_] -> ()`
g) `\x y x -> y + 2 * x`
h) `(\x y -> x y) (\x y -> x y) (\x y -> x y)`
i) `\a b -> a (\c d e -> b c (d e))`
j) `\[] -> ()`

Příklad 4.1.2 Jsou následující úpravy korektní?

a) `\x -> (<x)` ⇝ `\x -> flip x (<)`
b) `\x -> (.) f (g x)` ⇝ `\x -> (.) (f . g) x`
c) `f . (.g)` ⇝ `\x -> f (.g x)`
d) `\x y z -> const (+) x y z` ⇝ `\x y z -> (+) y z`
e) `_ -> (+3) 2` ⇝ `_ -> 2 + 3`

Příklad 4.1.3 Jaký je rozdíl mezi následujícími funkcemi?

- `\x y z -> 10 * x - mod y 4`
- `\x y -> \z -> 10 * x - mod y 4`
- `\x -> \y z -> 10 * x - mod y 4`
- `\x -> \y -> \z -> 10 * x - mod y 4`

Příklad 4.1.4 V následujících výrazech proveďte aplikace λ-abstrakcí na hodnoty tam, kde to je možné. Funkce přitom nevyhodnocujte -- zaměřte se skutečně jenom na aplikaci λ-abstrakcí. Také nepoužívejte η-redukci.

a) `(\t -> map t [1, 2, 3]) (+)`
b) `\t u -> t * u 2 3`
c) `\t u -> (t * u 2) 3`
d) `(\x y -> x + y) 4 0.3`
e) `(\n -> n * 10) (3 + 1)`
f) `(\f -> const map f filter) (head . head)`
g) `(\a b -> zipWith a [1..10] b) (\x y -> x * 10 + y) ((\t -> map (^2) t) [1..5])`
h) `(\x y -> x (map y)) (\s (a, b) -> s [a..b]) (\f -> f - 1)`

4.2 η-redukce, pointfree vs. pointwise zápis

Příklad 4.2.1 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 -> 3 + 60 `div` x ^ 2 > 0``
d) `\x -> [x]`
e) `\s -> "<" ++ s ++ ">"`
f) `\x -> 0 < 35 - 3 * 2 ^ x`
g) `\x y -> x ^ y`
h) `\x y -> y ^ x`
i) `\x y -> 2 * x + y`

Příklad 4.2.2 Převeďte následující funkce do pointfree tvaru:

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

Příklad 4.2.3 Převeďte následující výrazy do pointwise tvaru:

a) `(^2) . mod 4 . (+1)`
b) `(+) . sum . take 10`
c) `map f . flip zip [1, 2, 3]`
d) `(.)`
e) `flip flip 0`
f) `(.) (+) . (+)`
g) `(.(.))`

Příklad 4.2.4 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 4.2.5 Zjistěte, co dělají následující funkce a určete jejich typ:

a) `h1 = (.(,)) . (.) . (,)`
b) `h2 = ((,).) . (,)`

Příklad 4.2.6 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 4.2.7 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 4.2.8 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 4.2.9 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 Curryfikace

Příklad 4.3.1 Definujte *unární* funkci `nebo` pro realizaci logické disjunkce a pomocí modifikátorů `curry` a `uncurry` definujte ekvivalenci mezi vámi definovanou funkcí `nebo` a předdefinovanou funkcí `(||)`.

Příklad 4.3.2 Analogicky k funkcím `curry` a `uncurry` definujte funkce

a) `curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d`
b) `uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d`

Příklad 4.3.3 Lze funkce `curry3, uncurry3` vyjádřit pomocí funkcí `curry, uncurry`?

Příklad 4.3.4 Převeďte funkce do pointfree tvaru:

a) `\(x, y) -> x + y`
b) `\x y -> nebo (x, y)` (`nebo = uncurry (||)`)
c) `\((x, y), z) -> x + y + z` (dodržte asociativitu operátoru +)

Příklad 4.3.5 Zaveďme funkci `dist f g x = f x (g x)`.

a) Vyjádřete funkci `dist (curry id) id` pomocí λ-abstrakce.
b) Co dělá funkce `pair = uncurry (dist . ((.) (curry id))`

Cvičení 5

5.1 Akumulační funkce na seznamech

Příklad 5.1.1 Definujte následující funkce rekurzivně:

a) `product'` -- součin prvků seznamu
b) `length'` -- počet prvků seznamu
c) `map'` -- funkci `map`

Co mají tyto definice společné? Jak by vypadalo jejich zobecnění?

Příklad 5.1.2 Určete, co dělají akumulační funkce s uvedenými argumenty. Najděte hodnoty, na které je lze tyto výrazy aplikovat, a ověřte pomocí interpretu.

a) `foldr (+) 0`
b) `foldr1 (\x s -> x + 10 * s)`
c) `foldl1 (\s x -> 10 * s + x)`

Příklad 5.1.3 Definujte funkci `subtractlist`, která odečte druhý a všechny další prvky *neprázdného* seznamu od jeho prvního prvku, tj. `subtractlist` $[x_1, \ldots, x_n] = x_1 - x_2 - \ldots - x_n$.

Příklad 5.1.4 Předpokládejme, že funkce `compose` skládá všechny funkce ze seznamu funkcí, tj. `compose` $[f_1, \ldots, f_n] = f_1 \; . \; \ldots \; . \; f_n$.

a) Definujte funkci `compose` s využitím akumulátorových funkcí.
b) Jaký je typ funkce `compose`? (Odvoďte bez použití interpretru a poté ověřte.)

Příklad 5.1.5 Uvažme funkci: `foldr (.) id`

a) Jaký je význam uvažované 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.

Příklad 5.1.6 Jaký je význam a typ funkce `foldr (:)`?

Příklad 5.1.7 Jaký je význam a typ funkce `foldl (flip (:)) []`?

Příklad 5.1.8 Vaší úlohou je implementovat funkci dle specifikace v zadání za použití standardních funkcí `foldr`, `foldl`, `foldr1`, `foldl1`. Požadovaný typ funkce je vždy uveden. 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.

a) Funkce `lengthFold` vrátí délku zadaného seznamu.

```
lengthFold :: [a] -> Int
lengthFold [1,2,3,5,8,0] ⇝* 6
```

b) Funkce `sumFold` vrátí součet čísel v zadaném seznamu.

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

c) Funkce `productFold` vrátí součin čísel v zadaném seznamu.

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

d) Funkce `orFold` vrátí `True`, pokud se v zadaném seznamu nachází aspoň jednou hodnota `True`, jinak vrátí `False`.

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

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

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

f) Funkce `minimumFold` vrátí minimální prvek ze zadaného neprázdného seznamu.

```
minimumFold :: Ord a => [a] -> a
minimumFold ['f','e','r','t'] ⇝* 'e'
```

g) Funkce `maximumFold` vrátí maximální prvek ze zadaného neprázdného seznamu.

```
maximumFold :: Ord a => [a] -> a
maximumFold ['f','e','r','t'] ⇝* 't'
```

h) Funkce `maxminFold` vrátí minimální a maximální prvek ze zadaného seznamu ve formě uspořádané dvojice. Použijte i formální argument seznam, budete ho potřebovat při definici.

```
maxminFold :: Ord a => [a] -> (a,a)
maxminFold [1,5,7,2,6,8,2] ⇝* (1,8)
```

i) 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 [(*8),(+2),(flip mod 5)] 27 ⇝* 32
```

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

```
idFold :: [a] -> [a]
idFold [5,3,2,1,4,1] ⇝* [5,3,2,1,4,1]
```

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

```
concatFold :: [[a]] -> [a]
concatFold [[1,2],[3],[4,5]] ⇝* [1,2,3,4,5]
```

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

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

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

```
nullFold :: [a] -> Bool
nullFold [1,2,5,6] ⇝* False
```

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

```
headFold :: [a] -> a
headFold [2,1,5,4,8] ⇝* 2
```

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

```
lastFold :: [a] -> a
lastFold [2,1,5,7,8] ⇝* 8
```

p) Funkce `reverseFold` vrátí zadaný seznam s prvky v obráceném pořadí.

```
reverseFold :: [a] -> [a]
reverseFold "asdfghj" ⇝* "jhgfdsa"
```

q) 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",""]
```

r) 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]
```

s) 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]
```

t) 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])
```

u) 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]
```

v) 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]
```

Příklad 5.1.9 Definujte funkci `foldl` pomocí funkce `foldr`.

Příklad 5.1.10 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 `f` pro použití ve výrazu `snd $ foldl f (True, []) s`?

Příklad 5.1.11 Proč je implementace funkce `or` (logická disjunkce všech hodnot v seznamu) pomocí funkce `foldr` lepší než pomocí `foldl`?

Příklad 5.1.12 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.

5.2 Líné vyhodnocování a práce s nekonečnými seznamy

Příklad 5.2.1 Uvažte význam líného vyhodnocování v následujících výrazech:

a) `take 10 [1..]`
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 5.2.2 Definujte funkce `cycle` a `replicate` pomocí jednodušších funkcí (lze při tom použít funkci `repeat`).

Příklad 5.2.3 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 sudých mocnin čísla 3.
d) Rostoucí seznam všech lichých mocnin čísla 3.
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 5.2.4 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 5.2.5 Pomocí rekurzivní definice a funkce `zipWith` vyjádřete Fibonacciho posloupnost.

Příklad 5.2.6 Elegantním způsobem zapište nekonečný výraz $p = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots$

Cvičení 6

6.1 Úvod do IO

Příklad 6.1.1 Do interpretru zapište nejprve výraz `"**\n**\n"` a poté výraz `putStr "**\n**\n"`. Rozdíl chování interpretru vysvětlete.

Příklad 6.1.2 Uvažme seznam definovaný následovně:

```
pt :: [[Integer]]
pt = iterate (\r -> zipWith (+) ([0] ++ r) (r ++ [0])) [1]
```

a) Vypište jeho prvních 15 prvků.

b) Co seznam vyjadřuje?

c) Vyhodnoťte `take 7 pt`.

d) Vyhodnoťte `show (take 7 pt)`.

e) Vyhodnoťte `map show (take 7 pt)`.

f) Vyhodnoťte `(unlines . map show) (take 7 pt)`.

g) Vyhodnoťte `(putStr . unlines . map show) (take 7 pt)`.

Příklad 6.1.3 Z níže uvedené URL stáhněte program `pt.hs`, spusťte ho a pochopte, jak funguje. Program je možné stáhnout z následující URL:
`https://is.muni.cz/auth/el/1433/podzim2014/IB015/um/seminars/code/pt.hs` nebo jej najdete v příloze A.1.

6.2 IO pomocí do-notace

Příklad 6.2.1 Definujte akci `getInt :: IO Int`, která ze standardního vstupu načte celé číslo. Využijte knihovní funkci `read :: (Read a) => String -> a`.

Příklad 6.2.2 Upravte a doplňte následující zdrojový kód tak, aby program vyžadoval a načetl postupně tři celá čísla a o nich určoval, zda mohou být délkami hran trojúhelníku. Hotový program přeložte do samostatného spustitelného souboru a otestujte.

```
main :: IO ()
main = do putStrLn "Enter one number:"
          x <- getLine
          print ((+) 1 (read x :: Int))
```

Příklad 6.2.3 Napište program, který vyzve uživatele, aby zadal jméno souboru, poté ověří, že zadaný soubor existuje, a pokud ano, vypíše jeho obsah na obrazovku, pokud ne, informuje o tom uživatele. Úkol řešte s využitím `doesFileExist` z modulu `System.Directory`

Příklad 6.2.4 Vysvětlete význam a rizika rekurzivního použití volání funkce `main` v následujícím programu.

```
main :: IO ()
main = do putStr "Enter string: "
          s <- getLine
          if null s then putStrLn "pa pa"
                    else do putStrLn (reverse s)
                            main
```

6.3 IO pomocí operátorů >>= a >>

Příklad 6.3.1 Uvažme následující program:

```
import Data.Char
main :: IO ()
main = getLine >>= putStr . filter isAlpha
```

a) Co program dělá?
b) Přepište program do `do`-notace.

Příklad 6.3.2 Převeďte následující program v `do`-notaci na notaci s použitím `>>=`.

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

Příklad 6.3.3 Které z následujících výrazů jsou korektní?

a) `getLine (>>=) putStrLn . tail`
b) `(>>) getLine getLine`
c) `getLine >>= \s -> return >> putStrLn "ok"`
d) `getLine >> getLine >>= (\a b -> putStrLn (a ++ b))`
e) `readFile "/etc/passwd" >>= (\s -> writeFile "backup") >> putStrLn s`
f) `getLine >>= \f -> putStrLn "N/A"`
g) `getLine >> _ -> return 1`
h) `x <- getChar >> putStrLn x >> putStr "done"`

Příklad 6.3.4 Následující funkci přepište do tvaru, ve kterém nepoužijete konstrukci `do`, také určete typ funkce.

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

Příklad 6.3.5 Funkci `query` z předchozího příkladu modifikujte 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.

Příklad 6.3.6 Upravte program `guess.hs` tak, aby parametry funkce `guess` četl z příkazové řádky. Program je možné stáhnout z následující URL:
https://is.muni.cz/auth/el/1433/podzim2014/IB015/um/seminars/code/guess.hs nebo jej najdete v příloze A.2.

Příklad 6.3.7 Vymyslete a naprogramujte několik triviálních prográmků manipulujících se soubory. Definice alternativně přepište s a bez pomoci syntaktické konstrukce `do`.

Příklad 6.3.8 Definujte funkci `(>>)` pomocí funkce `(>>=)`.

6.4 Vlastní datové typy

Příklad 6.4.1 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.

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

Příklad 6.4.2 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).

Příklad 6.4.3 Identifikujte nově vytvořené typové a datové konstruktory a určete jejich aritu.

a) `data X = X`
b) `data A = X | Y String | Z Int Int`
c) `data B a = A | B a | C a`
d) `data C = D C`
e) `data E = E (E, E)`
f) `type String = [Char]`

Příklad 6.4.4 Identifikujte nově vytvořené typové a datové konstruktory a určete jejich aritu.

a) `data X = Value Int`
b) `data X a = V a`
c) `data X = Test Int [Int] X`
d) `data X = X`
e) `data M = A | B | N M`
   `data N = C | D | M N`
f) `data Test a = F [Test] a | M Int (Maybe String) deriving Read`

g) `data Ha = Hah Int Float [Hah]`
h) `data FMN = T (Int, Int) (Int -> Int) [Int]`
i) `data LInt = [Int]`
j) `type Fat = Float -> Float -> Float`

Příklad 6.4.5 Mějme následující definici:

```
data Teleso = Kvadr Float Float Float   -- a,b,c
            | Valec Float Float         -- r,v
            | Kuzel Float Float         -- r,v
            | Koule Float               -- r
```

a) Jaké hodnoty má typ `Teleso`?
b) Kolik je v definici použito datových konstruktorů a které to jsou?
c) Kolik je v definici použito typových konstruktorů a které to jsou?
d) Definujte funkce `objem` a `povrch`, které pro hodnoty uvedeného typu počítají požadované.
e) Rozšiřte uvedený datový typ o další konstruktory a upravte odpovídajícím způsobem vámi definované funkce.

Příklad 6.4.6 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.

Příklad 6.4.7 Rozšiřte definici z předchozího příkladu o nulární datový konstruktor `X`, který bude zastupovat *proměnnou* tak, jak známe z aritmetických výrazů, tj.

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

[...??]

Příklad 6.4.8 Uvažte datový typ `type Frac = (Int, Int)`, kde hodnota `(a, b)` představuje zlomek $\frac{a}{b}$ (můžete předpokládat $b \neq 0$). Napište funkci nad datovým typem `Frac`, která

a) převede zlomek do základního tvaru;
b) zjistí, jestli se zadané dva zlomky rovnají;
c) vrátí `True`, jestli zlomek představuje nezáporné číslo;
d) vypočítá součet dvou zlomků;
e) vypočítá rozdíl dvou zlomků;
f) vypočítá součin dvou zlomků;
g) vypočítá podíl dvou zlomků (ověřte, že druhý zlomek je nenulový);
h) vrátí aritmetický průměr zadaného seznamu zlomků (opět ve formě zlomku).

Ve všech případech vraťte výsledek v základním tvaru.

Příklad 6.4.9 Které deklarace datových typů jsou správné?

a) `data M a = M a`

b) `data MujBool = Bool`
c) `type MujBool = Bool`
d) `data N x = NVal (x -> x)`
e) `type F = Bool -> Bool`
f) `type Makro = a -> a`
g) `data M = N (x, x) | N Bool | O M`
h) `type Fun a = a -> (a, Bool) -> c`
i) `type Fun (a, c) (a, b) = (b, c)`
j) `type Discarder a b c d e = b`
k) `data F = X Int | Y Float | Z X`
l) `data F = X Int | Y Float | Z (X Int)`
m) `data F = intfun Int`
n) `data F = Makro Int -> Int`
o) `type Val = Int | Bool`
p) `data M = Value M`
q)
```
data X1 = M Int
data X2 = M Float X1 | None
```
r)
```
data Choice x y = GoodChoice x | BadChoice y
type GC x = GoodChoice x
```
s)
```
data M1 = M1Val M2 | E
data M2 = M2Val M1
```
t) `data X = X X X`

Příklad 6.4.10 Určete typy následujících hodnot:

```
data T a = F String a | D String Int [T a]
```

a) `F String`
b) `D "abc" 2 [F "1" 1, F "2" 10]`
c) `D "main" 10 [F "Ano" "hello.c", F "Nie" "hello.o"]`
d) `[D "n1" 0 [], D "n2" 1 [T "2"]]`

```
data A a = M (a, a) | N [a] [A a] Int | O
```

e) `M ('a', 'S')`
f) `O`
g) `N [] [O, M (3, 3)] 0`
h) `x = N [] (repeat x) 10`

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

i) `N (M (N A))`
j) `M (M C)`

```
data X a b = T (X a b) | U (X b a) | V a | W b
```

k) `V True`
l) `T`
m) `[U (V (Just 2)), T (V [2])]`
n) `T . U . V`
o) `T (V W)`

Cvičení 7

7.1 Typové třídy

Příklad 7.1.1 Vysvětlete, co je to typová třída a jak může programátorovi použití typové třídy pomoci (ušetřit práci) při tvorbě a definici vlastních typů.

Příklad 7.1.2 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`.

Příklad 7.1.3 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 definujte jako lexikografické po složkách. Zobrazování hodnot tohoto typu nechť je slovní (tedy namísto obligátního `(1,2)` vypište třeba `"pair of 1 and 2"`).

Příklad 7.1.4 Vytvořte speciální verzi podmíněného výrazu s názvem `iff`, který bude mít polymorfní podmínku (tedy nevyžaduje striktně typ `Bool`). Vytvořte si pomocnou typovou třídu `Boolable`, která bude sdružovat všechny typy, které se dají interpretovat jako `Bool`. Typová třída bude zaručovat implementaci funkce `getBool :: Boolable a => a -> Bool`. Vytvořte instance pro několik základních typů (`Bool`, `Int`, `[a]`, …).

Příklad 7.1.5 Jaký je rozdíl mezi těmito dvěma definicemi? Předpokládejte datový typ `Nat` zavedený jako `data Nat = Zero | Succ Nat`.

- ```
  instance Ord Nat where
    (<=) Zero (Succ _) = True
    ...
  ```
- ```
  (<=) Zero (Succ _) = True
  ...
  ```

7.2 Maybe

Příklad 7.2.1 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.

a) `Maybe (Just a)`

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 [] :: Maybe [Maybe Int]`
k) `(Just 3, Just Nothing) :: (Maybe Int, Maybe a)`
l) `Maybe [a -> Just Char]`
m) `Just (\x -> x^2)`
n) `(Just (+1)) (Just 10)`
o) `\b matters -> if b then Nothing else matters`
p) `Just`
q) `Just Just`
r) `Just Just Just`
s) `Maybe Maybe`
t) `Maybe ((Num a) => Maybe (a, a))`

Příklad 7.2.2 S využitím typového konstruktoru `Maybe` definujte funkci `divlist :: Integral a => [a] -> [a] -> [Maybe a]`, která celočíselně podělí dva celočíselné seznamy „po složkách“, tj.

$$\texttt{divlist [}x_1\texttt{, ..., }x_n\texttt{] [}y_1\texttt{, ..., }y_n\texttt{]} \rightsquigarrow^* \texttt{[div }x_1\ y_1\texttt{, ..., div }x_n\ y_n\texttt{]}$$

a ošetří případy dělení nulou.

Příklad 7.2.3 Uvažte následující datový typ:

```
data MyMaybe a = MyNothing
               | MyJust a
               deriving (Show, Read, Eq)
```

Deklarujte typ `MyMaybe` jako instanci typové třídy `Ord`. Za jakých podmínek může být typ `MyMaybe a` instancí třídy `Ord`?

7.3 Rekurzivní datové typy

Příklad 7.3.1 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 datového konstruktoru `Succ` v daném výrazu.

e) Jak byste pomocí datového typu `Nat` zapsali nekonečno?

Příklad 7.3.2 Uvažme následující rekurzivní typ představující binární strom s ohodnocenými uzly:

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

a) Nakreslete všechny tříuzlové stromy typu `BinTree ()` a zapište je pomocí datových 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?
d) Definujte funkci `size :: BinTree a -> Int`, která určí počet uzlů stromu.

Příklad 7.3.3 Uvažte následující rekurzivní datový typ představující binární strom s ohodnocenými uzly:

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

Definujte následující funkce nad binárními stromy:

a) `treeSize :: BinTree a -> Integer`, která spočítá počet uzlů ve stromě
b) `treeMax :: Ord a => BinTree a -> a`, která najde maximální hodnotu v uzlech stromu
c) `listTree :: BinTree a -> [a]`, která převede všechny hodnoty uzlů ve stromu do seznamu
d) `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě a vrátí ohodnocení na ní

Příklad 7.3.4 Pro datový typ `BinTree` 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 ohodnocené hodnotou `v`.
b) Definujte funkci `height :: BinTree a -> Int`, která určí výšku stromu.
c) Definujte funkci `treeZip :: BinTree a -> BinTree b -> BinTree (a,b)` jako analogii seznamové funkce `zip`.

Příklad 7.3.5 Uvažme datový typ `BinTree`.

a) Definujte funkci `treeRepeat :: a -> BinTree a` jako analogii seznamové funkce `repeat`. Funkce tedy vytvoří nekonečný strom, který má zadanou hodnotu v každém uzlu.
b) Pomocí funkce `treeRepeat` vyjádřete nekonečný binární strom `nilTree`, který má v každém uzlu prázdný seznam.
c) Definujte 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.

Příklad 7.3.6 Deklarujte typ `BinTree a` jako instanci typové třídy `Eq`. Instanci si napište sami (tj. nepoužívejte klauzuli `deriving`).

Příklad 7.3.7 Definujte nějaký binární strom, který má nekonečnou hloubku (zkuste to udělat různými způsoby).

Příklad 7.3.8 Uvažte typ n-árních stromů definovaný následovně:

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

Definujte následující:

a) funkci `ntreeSize :: NTree a -> Integer`, která spočítá počet uzlů ve stromě
b) funkci `ntreeSum :: Num a => NTree a -> a`, která sečte ohodnocení všech uzlů stromu
c) instance `Eq` a `Ord` pro `NTree a`
d) funkci `ntreeMap :: (a -> b) -> NTree a -> NTree b`, která bere funkci a strom, a aplikuje danou funkci na ohodnocení v každém uzlu:

```
ntreeMap (+1) (NTree 0 [NTree 1 [], NTree 41 []])
    ⇝* NTree 1 [NTree 2 [], NTree 42 []]
```

Cvičení 8

8.1 Intensionální seznamy

Příklad 8.1.1 Pomocí intensionálních seznamů definujte funkci `divisors`, která k zadanému přirozenému číslu vrátí seznam jeho kladných dělitelů.

Příklad 8.1.2 Intensionálním způsobem zapište následující výrazy:

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

Příklad 8.1.4 Intensionálním způsobem zapište následující seznamy nebo funkce:

a) `[1,4,9,...,k^2]` (pro pevně dané `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) `[[1],[2,2,2],[3,3,3,3,3],[4,4,4,4,4,4,4],...]` (hledejte vztah mezi číslem a počtem jeho výskytů)
g) `["z","yy","xxx",...,"aaa...aaa"]` (znak `a` se v posledním členu vyskytuje přesně 26krát)
h) následující seznam „2D matic“

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

Příklad 8.1.5 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 8.1.6 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 8.1.7 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 8.1.8 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 8.1.9 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.

8.2 Katamorfismy nad vlastními datovými typy

Příklad 8.2.1 Mějme datový typ `Nat` reprezentující přirozená čísla:

```
data Nat = Zero | Succ Zero
```

Definujte funkci `natfold`, která je tzv. katamorfismem na typu `Nat` (tj. je funkcí, která nahrazuje všechny datové konstruktory datového typu, stejně jako je akumulační funkce `foldr` seznamovým katamorfismem).

```
natfold :: (a -> a) -> a -> Nat -> a
```

Příklady zamýšleného použití funkce `nfold`:

1. Funkce `natfold (Succ . Succ) Zero :: Nat -> Nat` „zdvojnásobuje“ hodnotu typu `Nat`.

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

Příklad 8.2.2 Vaší úlohou je implementovat funkce dle zadaných specifikací s použitím funkce `treeFold` zadané níže. Požadovaný typ funkce je vždy uveden. Jestli úloha neříká jinak, řešení by mělo být bez formálních parametrů, tedy v následujícím tvaru:

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

Na pořadí zpracovávaní jednotlivých uzlů nezáleží, ideálně však zpracujte vždy nejdříve levý podstrom, pak samotný vrchol a na závěr pravý podstrom (v některých podúlohách se bude výsledek lišit dle pořadí zpracovávání).

Úlohy řešte pro binární stromy typu `BinTree a` používané na cvičení. Jejich definice spolu s definicí „foldovací“ funkce `treeFold` jsou pro úplnost uvedené níže.

```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
               deriving Show
treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold f e Empty = e
treeFold f e (Node v l r) = f v (treeFold f e l) (treeFold f e r)
```

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 úplný tvar najdete až za poslední podúlohou.

a) Funkce `treeSum` vrátí součet čísel ve všech uzlech zadaného stromu.

   `treeSum :: Num a => BinTree a -> a`
   `treeSum tree01` $\rightsquigarrow^*$ `16`

b) Funkce `treeProduct` vrátí součin čísel ve všech uzlech zadaného stromu.

   `treeProduct :: Num a => BinTree a -> a`
   `treeProduct tree01` $\rightsquigarrow^*$ `120`

c) Funkce `treeOr` vrátí `True`, jestli se v zadaném stromě nachází alespoň jedenkrát hodnota `True`, jinak vrátí `False`.

   `treeOr :: BinTree Bool -> Bool`
   `treeOr tree05` $\rightsquigarrow^*$ `True`

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

   `treeSize :: BinTree a -> Int`
   `treeSize tree01` $\rightsquigarrow^*$ `6`
   `treeSize tree06` $\rightsquigarrow^*$ `5`

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

   `treeHeight :: BinTree a -> Int`
   `treeHeight tree03` $\rightsquigarrow^*$ `2`
   `treeHeight tree01` $\rightsquigarrow^*$ `3`

f) 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"]
```

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

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

h) Funkce `treeMax` vrátí maximální hodnotu ze všech hodnot v uzlech. Hodnoty musí být z typové třídy `Ord` a `Bounded` (poznámka: zkuste funkce `minBound` a `maxBound`).

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

i) 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"
```

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

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

k) 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"]
```

l) 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
```

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

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

n) 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
```

o) 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"]
```

p) 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
treeSum (treeMap (+1) tree01) ⇝* 22
```

q) 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
```

r) 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
```

s) 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))
```

t) Funkce `findPredicates` vezme 2 argumenty: základní hodnotu a strom, který má v každém uzlu uspořádanou dvojici tvořenou „identifikačním“ číslem a predikátem. Úlohou je vrátit seznam čísel odpovídajících predikátům, které se na dané základní hodnotě vyhodnotí na `True`. Výsledná funkce může mít jeden formální parametr.

```
findPredicates :: a -> BinTree (Int, a -> Bool) -> [Int]
findPredicates 3 tree06 ⇝* [1,3,4]
findPredicates 6 tree06 ⇝* [0,4]
```

Ukázkové stromy:

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

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

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

tree04 :: BinTree a
```

```
tree04 = Empty

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

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

Cvičení 9

9.1 Opakování

Příklad 9.1.1 Otypujte následující výrazy:

a) `head [id, not]`
b) `\f -> f 42`
c) `\t x -> x + x > t x`
d) `\xs -> filter (> 2) xs`
e) `\f -> map f [1,2,3]`
f)
```
foo f True  = map f [1,2,3]
foo f False = filter f [1,2,3]
```
g) `\(p,q) z  ->  q (tail z) : p (head z)`

Příklad 9.1.2 Uvažte následující datový typ:

```
data Foo a = Bar [a]
           | Baz a
```

Určete typy následujících výrazů:

a)
```
getlist (Bar xs) = xs
getlist (Baz x ) = x : []
```
b) `\foo -> foldr (+) 0 (getlist foo)`

Příklad 9.1.3 Definujte funkci `minmax :: Ord a => [a] -> (a, a)`, která pro daný neprázdný seznam *v jednom průchodu* spočítá minimum i maximum.

Funkci definujte jednou rekurzivně a jednou pomocí `foldr` nebo `foldl` (ne `foldr1`, `foldl1`).

Dále implementujte funkci `minmaxBounded :: (Ord a, Bounded a) => [a] -> (a, a)`, která funguje i na prázdných seznamech. Využijte konstant `minBound :: Bounded a => a`, `maxBound :: Bounded a => a`.

Najděte datový typ, pro který `minmaxBounded` nefunguje.

Poznámka: Může se stát, že interpretr bude zmatený z požadavku `Bounded a` a nebude schopen sám vyhodnotit výrazy jako `minmaxBounded [1,2,3]`. V takovém případě explicitně určete typ prvků seznamu, například: `minmaxBounded [1,2,3::Int]`.

Příklad 9.1.4 Převeďte následující funkce do pointwise tvaru a přepište je s pomocí intensionálních seznamů a bez použití funkcí `map, filter, curry, uncurry, zip, zipWith`.

a) `map . uncurry`
b) `\f xs -> zipWith (curry f) xs xs`
c) ``map (* 2) . filter odd . map (* 3) . map (`div` 2)``
d) `map (\f -> f 5) . map (+)`

Příklad 9.1.5 Otypujte následující `IO` výrazy a převeďte je do `do`-notace: Uvažujte při tom následující typy `(>>=), (>>), return`:

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

a) `readFile "/etc/passwd" >> putStrLn "bla"`
b) `\f -> putStrLn "bla" >>= f`
c) `getLine >>= \x -> return (read x)`
d)
```
foo :: Integer
foo = getLine >>= \x -> read x
```

Příklad 9.1.6 Které z následujících výrazů jsou korektní?

a) `max a b + max (a - b) (b + a) (b * a)`
b) `False < True || True`
c) `5.1 ^ 3.2 ^ 2`
d) `2 ^ if even m then 1 else m`
e) `m mod 2 + 11 / m`
f) `(/) 3 2 + 2`
g) `[(4,5),(7,8),(1,2,3)]`
h) `(\s -> s, s)`
i) `[ x | x <- [1..10], odd x, let m = 5 * x - 1]`
j) `('a':"bcd", "a":"bcd", "a":"bcd":[])`
k) `[] : map f x`
l) `map f x : []`
m) `fst (map, 3) fst []`
n) `[[1],[2]..[10]]`
o) `\t -> if t == (x:_) then x else 0`
p) `\x s -> fst (x:s) : repeat x`
q) `[ [] | _ <- [] ]`
r) `[ m * n | m <- [1..] | n <- iterate (^2) 1 ]`
s) `zip [10,20..] [1,4,9,16,..]`
t) `getLine >>= putStrLn`
u) `getLine >> putStrLn`
v) `\t -> getLine >> putStrLn t`
w) `(>>) (putStrLn "OK") . putStrLn`
x) `x >>= (f >>= g) >>= h`

Příklad 9.1.7 Které z následujících typů jsou korektní?

a) `[Int, Int]`
b) `(Int)`
c) `[()] -> ([])`
d) `(Show x) => [x]`
e) `a -> b -> c -> d -> e`
f) `(A -> b) -> [A] -> b`
g) `String -> []`
h) `IO (String -> IO ())`
i) `IO (Maybe Int)`
j) `IO a -> IO a`

k) `[string] -> int -> bool -> string`
l) `Int a => b`
m) `a -> (Int b => b)`
n) `(Integral a, Num a) => a`
o) `Num a -> Num a`
p) `Num -> c -> c`

Příklad 9.1.8 Vyhodnoťte následující výrazy (zjednodušte do co nejjednoduššího tvaru).

a) `init [1] ++ [2]`
b) `head []`
c) `concat []`
d) `map f []`
e) `map (map (0:)) [[]]`
f) `map (++[]) [[[], []], [], [[]]]`
g) `[ 0 | False ]`
h) `[ [] | _ <- [[]] ]`
i) `[ [[]] | _ <- [] ]`
j) `(\x -> 3 * x + (\x -> x + x ^ 2) (2 * x - 1)) 5`
k) `(\f x -> f id (max 5) x) (.) 3`
l) `[] ++ map f (x ++ [])`

Příklad 9.1.9 Které z následujících úprav jsou korektní?

a) `(+1) (*2)` $\rightsquigarrow$ `(+1) . (*2)`
b) `f . (.g)` $\rightsquigarrow$ `\x -> f (.g x)`
c) `getLine >>= \x -> putStrLn (reverse x) >> putStrLn "done"` $\rightsquigarrow$
`(getLine) >>= (\x -> putStrLn (reverse x)) >> (putStrLn "done")`
d) `\x y -> (\z -> f z) + 3` $\rightsquigarrow$ `\x y z -> f z + 3`
e) `and (zipWith (==) s1 s2) && False` $\rightsquigarrow^*$ `False`

Příklad 9.1.10 Otypujte následující výrazy:

a) `[]`
b) `[()]`
c) `tail [True]`
d) `(id.)`
e) `flip id`
f) `id (id id (id id)) ((id id) id)`
g) `(.) (.)`
h) `map flip`
i) `(.) . ((.).)`
j) `\g -> g (:) []`
k) `\s -> [t | (h:t) <- s, h]`
l) `\x y -> (x y, y x)`
m) `\[] -> []`
n) `(>>getLine)`
o)
```
do  x <- getLine
    let y = reverse x
    putStrLn ("reverted: " ++ y)
```

Příklad 9.1.11 Definujte funkci `nth :: Int -> [a] -> [a]`, která vybere každý n-tý prvek ze seznamu (seznam začíná nultým prvkem, můžete předpokládat, že $n \geq 1$). Zkuste úlohu vyřešit několika způsoby. Příklad: `nth 3 [1..10]` $\rightsquigarrow^*$ `[1,4,7,10]`.

Příklad 9.1.12 Definujte funkci `modpwr`, která funguje stejně jako funkce `\n k m -> mod (n^k) m`, ale implementujte ji efektivně (tak, aby v průběhu výpočtu nevycházela čísla, která jsou větší než $\texttt{n}^3$). Můžete předpokládat, že `k` $\geq$ `0`, `m` $\geq$ `1`. Funkce by měla mít logaritmickou časovou složitost vzhledem na velikost `k`.

Pomůcka: Použijte techniku *exponentiation by squaring* a dělejte zbytky už z mezivýsledků.

Příklad 9.1.13 Co dělají následující funkce?

a) `f1 = flip id 0`
b) `f2 = flip (:) []`
c) `f3 = zipWith const`
d) `f4 p = if p then ('/':) else id`
e) `f5 = foldr id 0`
f) `f6 = foldr (const not) True`

Příklad 9.1.14 Které z následujících vztahů jsou *obecně* platné (tj. platí pro každou volbu argumentů)? Uvažte také typ výrazů. Neplatné vztahy zkuste opravit.

a) `reverse (s ++ [x])` $\equiv$ `x : reverse s`
b) `map f . filter p` $\equiv$ `filter p . map f`
c) `flip . flip` $\equiv$ `id`
d) `foldr f (foldr f z s) t` $\equiv$ `foldr f z (s ++ t)`
e) `sum (zipWith (+) m n)` $\equiv$ `sum m + sum n`
f) `head s : tail s` $\equiv$ `s`
g) `map f (iterate f x)` $\equiv$ `iterate f (f x)`
h) `foldr (1+) 0` $\equiv$ `length`

Příklad 9.1.15 V následujících výrazech doplňte vhodný podvýraz za ... tak, aby ekvivalence platily *obecně* (tedy pro libovolnou volbu proměnných).

a) `foldr ... z s` $\equiv$ `foldr f z (map g s)`
b) `zipWith ... x y` $\equiv$ `map f (zipWith g (map h1 x) (map h2 y))`
c) `foldl ... z s` $\equiv$ `foldl f z (filter p s)`
d) `foldr ... [] (s1, s2)` $\equiv$ `zip s1 s2`
e) `foldr ...` $\equiv$ `const :: a -> [b] -> a`

Příklad 9.1.16 Jakou časovou složitost má vyhodnocení následujících výrazů? Uvažujte normální redukční strategii. Určete jenom asymptotickou složitost (konstantní, lineární, kvadratická, ..., exponenciální, výpočet nekončí). Předpokládejte, že použité proměnné jsou již plně vyhodnoceny a jejich hodnotu lze získat v jednom kroku.

a) `head s` (vzhledem k délce `s`)
b) `sum s` (vzhledem k délce `s`, pro jednoduchost předpokládejte, že operace sčítaní má konstantní složitost)
c) `take m [1..]` (vzhledem k hodnotě `m`)
d) `take m [1..10^6]` (vzhledem k hodnotě `m`)
e) `take n [1..m]` (vzhledem k hodnotám `m`, `n`)

f) `m ++ n` (vzhledem k délkám seznamů `m`, `n`)
g) `repeat x` (vzhledem k celočíselné hodnotě `x`)
h) `head (head s : tail s)` (vzhledem k délce seznamu `s`)
i) `fst (n, sum [1.0..n] / n)` (vzhledem k hodnotě `n`)

9.2 Složitější příklady

Příklad 9.2.1 Vaší úlohou je napsat program na řešení Hanojských věží. Tento matematický hlavolam vypadá následovně: Máme tři kolíky (věže), očíslujme si je 1, 2, 3. Na začátku máme na kolíku 1 všech N disků ($N \geq 1$) s různými poloměry postavených tak, že největší disk je naspodu a nejmenší navrchu. Vaší úlohou je přemístit disky na kolík 2 tak, aby byly stejně seřazené. Během celého procesu však musíme dodržet 3 pravidla:

- Disky přemísťujeme po tazích po jednom.
- Tah spočívá v tom, že vezmeme kotouč, který je na vrcholu některé věže a položíme ho na vrchol jiné věže.
- Je zakázáno položit větší kotouč na menší.

Napište funkci, která dostane počet disků, číslo kolíku, na kterém se na začátku disky nachází, a číslo kolíku, kam je chceme přesunout. Funkce vrátí seznam uspořádaných dvojic `(a,b)` -- tahů, kde `a` je číslo kolíku, ze kterého jsme přesunuli vrchní disk na kolík `b`. Tedy například `hanoi 3 1 2` vrátí `[(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2)]`.

Příklad 9.2.2 Dokažte, že funkce `($) . ($) . ... . ($)` představuje pro libovolný konečný počet `($)` vždy tu stejnou funkci. Zkuste intuitivně argumentovat, proč tomu tak je.

Příklad 9.2.3 Z funkcí nacházejících se v modulu `Prelude` vytvořte bez použití podmíněné konstrukce (`if`) funkci `if'` splňující `if' b x y = if b then x else y`

Příklad 9.2.4 Zkuste přepsat následující výraz pomocí intensionálních seznamů a funkcí pracujících se seznamy:

```
if cond1 then val1 else if cond2 then val2 else ... else val_default
```

Příklad 9.2.5 Co nejpřesněji popište, co dělají následující funkce.

a) `\s -> zipWith id (map map [even, (/=0) . flip mod 7]) (repeat s)`
b) `(\a b t -> (a t, b t)) (uncurry (flip const)) (uncurry const))`
c)
```
let g k 0 = k 1
    g k n = g (k . (n*)) (n - 1)
in f = g id
```
d) `f n = foldr (\k -> concat . replicate k) [0] [n,n - 1..1]`
e)
```
skipping = skip' id where
    {skip' f [x] = [f []]; skip' f (x:xs) = f xs : skip' (f . (x:)) xs}
```
f)
```
let f = 0 : zipWith (+) f g
    g = 1 : zipWith (+) (repeat 2) g}
in f
```

g) `boolseqs = []:[b:bs | bs <- boolseqs, b <- [False, True]]`

h)
```
let f = 1 : 1 : zipWith (tail g) g
    g = 1 : 1 : zipWith (tail f) f
in f
```

Příklad 9.2.6 Napište funkci `find :: [String] -> String -> [String]`, která vrátí všechny řetězce ze seznamu v prvním argumentu, které jsou podřetězcem řetězce v druhém argumentu. Předpokládejte, že všechny řetězce v prvním argumentu jsou neprázdné. Příklad:

`find ["a", "b", "c", "d"] "acegi"` $\rightsquigarrow^*$ `["a", "c"]`
`find ["1", "22", "321", "111", "32211"] "32211"` $\rightsquigarrow^*$ `["1", "22", "32211"]`

Příklad 9.2.7 Které z následujících funkcí není možno v Haskellu definovat (uvažujte standard Haskell98)?

a) `f x = x x`
b) `f = \x -> (\y -> x - (\x -> x * x + 2) y)`
c) `f k = tail . tail . ... . tail` (funkce se opakuje `k`-krát)
d) `f k = head . head . ... . head` (funkce se opakuje `k`-krát)

Příklad 9.2.8 Mějme dány výrazy `x, y`, které mají kompatibilní typ (lze je tedy unifikovat). Navrhněte nový výraz, který bez použití explicitního otypování vynutí, aby `x, y` měli stejný typ. Zkuste najít více řešení.

Příklad 9.2.9 Doplňte do výrazu `foldr f [1,1] (replicate 8 ())` vhodnou funkci místo `f` tak, aby se tento výraz vyhodnotil na prvních deset Fibonacciho čísel v klesajícím pořadí.

Příklad 9.2.10 Určete typ funkce `unfold` definované níže.

```
unfold p h t x = if p x then [] else h x : unfold p h t (t x)
```

Tato funkce intuitivně funguje jako seznamový *anamorfismus* (opak seznamového *katamorfismu* `foldr`). Tedy zatímco katamorfismus zpracovává prvky seznamu a vygeneruje hodnotu, anamorfismus na základě několika daných hodnot seznam vytváří.

Pomocí funkce `unfold` definujte následující funkce:

a) `map`
b) `filter`
c) `foldr`
d) `iterate`
e) `repeat`
f) `replicate`
g) `take`
h) `list_id` (tj. `id :: [a] -> [a]`)
i) `enumFrom`
j) `enumFromTo`

Nejvíce vnější funkcí by měla být funkce `unfold`.

Příklad 9.2.11 Popište množinu funkcí $\{dot_k \mid k \geq 1\}$, kde dot_k = `(.) (.) ... (.)` (k-krát).

Pomůcka: Při zjišťování vlastností funkce dot_k je možné využít zjištěné vlastnosti funkce dot_{k-1}.

Příklad 9.2.12 Je možné jen s pomocí funkce `id` a aplikace vytvořit nekonečně mnoho různých funkcí? A pomocí funkce `const`? (Připomínáme, že skládání je funkce a nemůžete ji tedy použít!)

Příklad 9.2.13 Najděte všechny plně definované konečné seznamy `s` (tj. pro každý jejich prvek platí, že jej lze v konečném čase vyhodnotit), pro které platí:

$\forall$ `f :: t -> t`. $\forall$ `p :: t -> Bool`. `filter p (map f s)` $\equiv$ `map f (filter p s)`

Uvažujte, že `f`, `p` jsou plně definované pro každý argument.

Cvičení 10

10.1 Úvod do Prologu, backtracking

Příklad 10.1.1 Spusťte a ukončete interpret jazyka Prolog.

Příklad 10.1.2 Načtěte rodokmen ze souboru *pedigree.pl* do prostředí Prologu. Soubor naleznete v ISu a v příloze sbírky. Formulujte vhodné dotazy a pomocí interpretru zjistěte odpovědi na následující otázky:

a) Je Petr rodičem Lenky?
b) Je Petr rodičem Jana?
c) Jaké děti má Pavla?
d) Má Petr dceru?
e) Kdo je otcem Petra?
f) Které dvojice otec-syn známe?

Příklad 10.1.3 Pro rodokmen ze souboru *pedigree.pl* naprogramujte následující predikáty:

a) `child/2`, který uspěje, jestliže první argument je dítětem druhého.
b) `grandmother/2`, který uspěje, jestliže první argument je babičkou druhého.
c) `stepBrother/2`, který uspěje, jestliže první argument je nevlastním bratrem druhého argumentu (mají tedy právě jednoho společného rodiče).

Příklad 10.1.4 Pro rodokmen ze souboru *pedigree.pl* napište predikát `descendant/2`, který uspěje, když je první argument potomkem druhého (ne nutně přímý). Bez použití interpretru určete, v jakém pořadí budou nalezeni potomci Pavly, když použijeme dotaz `?- descendant(X,pavla)`. Jaký vliv má pořadí klauzulí a cílů v predikátu `descendant` na jeho funkci?

Příklad 10.1.5 Namodelujte osoby a rodičovské vztahy relacemi `daughter/2` a `son/2` v rodině Simpsonových.

Naprogramujte predikát `sister/2`, který uspěje, pokud osoba zadaná jako druhý parametr je sestrou osoby zadané jako první parametr (nikdo si není sám sobě sourozencem). Zformulujte dotaz, jehož úplným vyhodnocením se dozvíte všechny Bartovy sestry. Zkuste stejným predikátem zjistit, jaké sestry má Lisa.

10.2 Unifikace

Příklad 10.2.1 Které unifikace uspějí, které ne a proč? Jaká substituce je výsledkem provedených unifikací?

a) `a(X) = b(X)`
b) `X = a(Y)`
c) `a(X) = a(X,X)`
d) `X = a(X)`

e) `jmeno(X,X) = jmeno(Petr,plus)`
f) `s(1,a(X,q(w))) = s(Y,a(2,Z))`
g) `s(1,a(X,q(X))) = s(W,a(Z,Z))`
h) `X = Y, P = R, s(1,a(P,q(R))) = s(Z,a(X,Y))`

Příklad 10.2.2 Pro každou z následujících unifikací rozhodněte, jestli uspěje. Jestli ano, zapište výslední substituci.

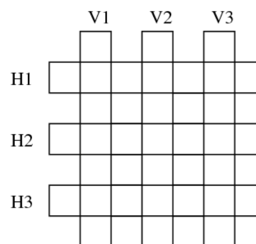
a) `monday = monday`
b) `'Monday' = monday`
c) `'monday' = monday`
d) `Monday = monday`
e) `monday = tuesday`
f) `day(monday) = monday`
g) `day(monday) = X`
h) `day(X) = day(monday)`
i) `day(monday,X) = day(Y,tuesday)`
j) `day(monday,X,wednesday) = day(Y,tuesday,X)`
k) `day(monday,X,wednesday) = day(Y,thursday)`
l) `day(D) = D`
m) `weekend(day(saturday),day(sunday)) = weekend(X,Y)`
n) `weekend(day(saturday),X) = weekend(X,day(sunday))`

Příklad 10.2.3 Unifikujte následující výrazy a zapište výslednou substituci.

a) `p(X)` a `p(f(Y))`
b) `p(X,f(X))` a `p(f(X),Y)`
c) `p(X,Y)` a `p(Z,Z)`
d) `p(X,8)` a `p(Y,Y)`
e) `p(f(X))` a `p(Z,Y)`
f) `p(f(Z),g(X))` a `p(Y,g(Y))`
g) `p(X,g(Z),X)` a `p(f(Y),Y,W)`

Příklad 10.2.4 Pro níže uvedenou databázi slov napište predikát `crossword/6`, který vypočítá, jak vyplnit uvedenou křížovku. První tři argumenty představují slova uváděná vertikálně shora dolů, druhé tři argumenty představují slova uváděná horizontálně zleva doprava.

```
word(astante,  a,s,t,a,n,t,e).
word(astoria,  a,s,t,o,r,i,a).
word(baratto,  b,a,r,a,t,t,o).
word(cobalto,  c,o,b,a,l,t,o).
word(pistola,  p,i,s,t,o,l,a).
word(statale,  s,t,a,t,a,l,e).
```



Příklad 10.2.5 Napište predikát `geq/2`, který jako argumenty dostane dvě čísla zadaná pomocí následníků a uspěje, pokud první číslo je větší nebo rovné druhému. Například dotaz `?- geq(s(0), s(s(0))).` neuspěje.

Příklad 10.2.6 Napište predikát `add/3`, který sčítá dvě přirozené čísla zadaná pomocí následníků. Například `?- add(s(0),s(s(0)),X).` uspěje se substitucí `X = s(s(s(0)))`.

Příklad 10.2.7 Napište predikát `mult/3`, který vypočítá součin dvou čísel zadaných pomocí následníků. V případě potřeby použijte predikát `add/3` definovaný v úloze 10.2.6. Příklad: dotaz `?- mult(s(s(0)),s(s(0)),X).` uspěje se substitucí `X = s(s(s(s(0))))`.

10.3 Backtracking a SLD stromy

Příklad 10.3.1 Uvažme následující program a dotaz `?- a.` Nakreslete odpovídající výpočetní SLD strom.

```
a :- b,c.
a :- d.
b.
b :- d.
c.
d :- e.
d.
```

Příklad 10.3.2 Uvažme následující databázi faktů:

```
r(a,b).
r(a,c).
r(b,d).

f1(a).
f1(X) :- f1(Y), r(Y,X).

f2(X) :- f2(Y), r(Y,X).
f2(a).

g1(a).
g1(X) :- r(Y,X), g1(Y).

g2(X) :- r(Y,X), g2(Y).
g2(a).
```

Zdůvodněte chování interpretru pro následující dotazy a nakreslete odpovídající SLD stromy.

1. `?- f1(X).`
2. `?- f2(X).`
3. `?- g1(X).`
4. `?- g2(X).`

Příklad 10.3.3 Uvažte následující program a dotaz `?- p(X,X).` Nakreslete odpovídající výpočetní SLD strom.

```
p(X,Y) :- q(X,Z), r(Z,Y).
p(X,X) :- s(X).
q(X,b).
q(b,a).
q(X,a) :- r(a,X).
r(b,a).
```

```
s(X) :- t(X,a).
s(X) :- t(X,b).
s(X) :- t(X,X).
t(a,b).
t(b,a).
```

Cvičení 11

11.1 Číselné operace

Příklad 11.1.1 Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 = 1 + 1`
b) `1 + 1 = 1 + 1`
c) `2 is 1 + 1`
d) `1 + 1 is 1 + 1`
e) `X = 1 + 1`
f) `X is 1 + 1`
g) `2 is 1 + X`
h) `X = 1, 2 is 1 + X`

Příklad 11.1.2 Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `X = 2`
b) `X == 2`
c) `X = 2, X == 2`
d) `X =:= 2`
e) `1 + 1 == 2`
f) `1 + 1 =:= 2`
g) `2 =:= 1 + 1`

Příklad 11.1.3 Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 < 2 + 1`
b) `1 + 2 =< 2 + 1`
c) `1 + 2 >= 1`
d) `2 * 3 > 3 * 1.5`
e) `1 + 1 \== 2`
f) `1 + 1 =\= 2`
g) `1 + 2 =\= 2`

Příklad 11.1.4 Jak se liší následující výrazy? Které by v případě dotazu uspěly, které ne, a které nejsou syntakticky správně? Při korektních unifikacích určete i výslednou substituci.

a) `X = Y + 1`
b) `X is Y + 1`
c) `X = Y`
d) `X == Y`
e) `1 + 1 = 2`
f) `2 = 1 + 1`
g) `1 + 1 = 1 + 1`
h) `2 is 1 + 1`
i) `1 + 1 is 1 + 1`
j) `1 + 2 =:= 2 + 1`

k) `X \== Y`
l) `X =\= Y`
m) `1 + 2 =\= 1 - 2`
n) `1 <= 2`
o) `1 =< 2`
p) `sin(X) is sin(2)`
q) `sin(X) = sin(2 + Y)`
r) `sin(X) =:= sin(2 + Y)`

Příklad 11.1.5 Rozhodněte, které z následujících dotazů uspějí a které ne.

a) `?- 15 is 3 * 5.`
b) `?- 14 =\= 3 * 5.`
c) `?- 15 = 3 * 5.`
d) `?- 15 == 3 * 5.`
e) `?- 15 =\= 3 * 5.`
f) `?- 15 =:= 3 * 5.`
g) `?- 3 * 5 == 3 * 5.`
h) `?- 3 * 5 == 5 * 3.`
i) `?- 3 * 5 =:= 3 * 5.`
j) `?- 10 - 3 =\= 9 - 2.`

Příklad 11.1.6 Napište predikát `convert/2`, který převede svůj první argument (přirozené číslo) na reprezentaci daného čísla pomocí následníků. Například `?- convert(3, X).` uspěje s unifikací `X = s(s(s(0)))`.

Příklad 11.1.7 Napište predikát `firstnums/2`, který spočítá součet prvních `N` přirozených čísel. Například dotaz `?- firstnums(5, X).` uspěje s unifikací `X = 15`.

Příklad 11.1.8 Implementujte predikát `fact/2`, který při dotazu `fact(`m`, `n`)` uspěje, pokud $m > 0$ a $n = m!$. V ostatních případech může interpret cyklit.

Predikát by měl rovněž fungovat při volání ve tvaru `fact(`n`, Res)`, kde `Res` je volná proměnná, a unifikovat tuto proměnnou s $n!$.

Příklad 11.1.9 Napište predikát `powertwo/1`, který uspěje, pouze když číslo zadané jako argument je mocninou dvou. Využijte fakt, že mocniny dvou lze opakovaně beze zbytku dělit dvěma, dokud nedostaneme jedna. Můžete využít binární predikáty `mod` pro modulo a `//` pro celočíselné dělení.

Příklad 11.1.10 Naprogramujte predikát `prime/1`, který uspěje, pokud `Num` je prvočíslo. K řešení lze použít naivní metodu testování zbytku po dělení všemi čísly od 2 po zadané číslo `Num`.

Příklad 11.1.11 Napište predikát `gcd/3`, který spočítá největšího společného dělitele dvou přirozených čísel. Například dotaz `?- gcd(21, 49, X).` uspěje s unifikací `X = 7`.

Využijte Euklidův algoritmus, který je založen na pozorování, že největší společný dělitel čísel a, b (kde $a > b$) je stejný jako největší společný dělitel čísel $(a - b)$, b.

Příklad 11.1.12 Napište predikát `dsum/2`, který spočítá ciferný součet zadaného přirozeného čísla. Například dotaz `?- dsum(12345, X).` uspěje s unifikací `X = 15`.

Příklad 11.1.13 Zamyslete se a zdůvodněte, proč následující logický program není vhodný, i když jeho výsledek je vždy korektní. Program počítá n-tý člen Fibonacciho posloupnosti přímo podle definice.

```
fib(0, 0).
fib(1, 1).
fib(N, X) :-
    N > 2, N1 is N - 1, N2 is N - 2, fib(N1, Y), fib(N2, Z), X is Y + Z.
```

11.2 Práce se seznamy

Příklad 11.2.1 Které z následujících zápisů představují korektní zápis seznamu? Pokud je zápis korektní, určete počet prvků v daném seznamu.

a) `[1|[2,3,4]]`
b) `[1,2,3|[]]`
c) `[1|2,3,4]`
d) `[1|[2|[3|[4]]]]`
e) `[1,2,3,4|[]]`
f) `[[]|[]]`
g) `[[1,2]|4]`
h) `[[1,2],[3,4]|[5,6,7]]`

Příklad 11.2.2 Implementujte vlastní verze predikátů, které pro seznamy počítají standardní seznamové funkce, které znáte z Haskellu. Konkrétně imitujte funkce `head`, `tail`, `last` a `init`.

Příklad 11.2.3 Napište následující predikáty pro práci se seznamy za pomoci vestavěného predikátu `append/3`.

a) `prefix/2` uspěje, jestliže je první argument prefixem seznamu ve druhém argumentu.
b) `suffix/2` uspěje, jestliže je první argument sufixem seznamu ve druhém argumentu.
c) `element/2` uspěje, jestliže je první argument členem seznamu ve druhém argumentu.
d) `adjacent/3` uspěje, jestliže jsou první dva prvky členy seznamu ve třetím argumentu a jsou vedle sebe (v daném pořadí).
e) `sublist/2` uspěje, jestliže je seznam v prvním argumentu podseznamem seznamu v druhém argumentu.

Příklad 11.2.4 Předpokládejme existenci predikátu `mf/2`, použitelného v módu (+,-). Vytvořte predikát `map/2`, který bude imitovat chování Haskellové funkce `map`, tj. bude možné pomocí něj „aplikovat funkci `mf`" na každý prvek zadaného seznamu.

Příklad 11.2.5 Určete, co počítá (jaký význam má) následující program:

```
something([], []).
something([H|T], [H|S]) :- delete(T, H, X), something(X, S).
```

Příklad 11.2.6 S využitím knihovních funkcí pro seznamy napište následující predikáty tak, aby je bylo možno používat v módu (+,?).

a) `variation3/2`, který uspěje, pokud je druhý argument tříprvkovou variací ze seznamu v prvním argumentu.
b) `combination3/2`, který uspěje, pokud je druhý argument tříprvkovou kombinací ze seznamu v prvním argumentu. Trojice prvků v kombinaci ať jsou uspořádány (lze k tomu využít binární operátor `@<`, který umožňuje porovnávat termy[1]).

Příklad 11.2.7 Napište predikát `doubles/2`, který uspěje, jestli prvky ve druhém seznamu jsou dvojnásobky prvků v prvním seznamu.

Například dotaz `?- doubles([5,3,2], [10,6,4]).` uspěje, zatímco dotaz `?- doubles([1,2,3], [2,4,5]).` neuspěje.

Příklad 11.2.8 Napište následující predikáty:

a) `listLength/2`, který vypočítá délku zadaného seznamu.
b) `listSum/2`, který vypočítá součet čísel v zadaném seznamu.
c) `fact/2`, který vypočítá faktoriál zadaného čísla.
d) `scalar/3`, který vypočítá skalární součin zadaných dvou seznamů (můžete předpokládat, že jsou stejné délky).

Zkuste tyto predikáty přepsat za pomocí akumulátoru, aby se při výpočtu mohla použít optimalizace posledního volání.

Příklad 11.2.9 Napište predikát `filter/2`, který ze seznamu odstraní všechny nečíselné hodnoty. Například dotaz `?- filter([5,s(0),3,a,2,A,7], X).` uspěje se substitucí `X = [5,3,2,7]`.

Příklad 11.2.10 Napište predikát `digits/2`, který ze seznamu cifer vytvoří číslo. Například dotaz `?- digits([2,8,0,7], X).` uspěje se substitucí `X = 2807`.

Příklad 11.2.11 Napište predikát `nth/3`, který vrátí n-tý prvek seznamu. Například dotaz `?- nth(4, [5,2,7,8,0], X).` uspěje se substitucí `X = 8`.

Příklad 11.2.12 Definujte následující predikáty s pomocí akumulátoru:

a) `listSum/2`, který sečte seznam čísel a na nečíselném seznamu neuspěje (použijte `number/1`),
b) `fact/2`, který spočítá faktoriál zadaného čísla,
c) `fib/2`, který (efektivně) spočítá n-tý člen Fibonacciho posloupnosti.

Příklad 11.2.13 Napište predikát `mean/2`, který spočítá aritmetický průměr zadaného seznamu. Například dotaz `?- mean([1,2,3,4,5,6], X).` uspěje se substitucí `X = 3.5`.

Příklad 11.2.14 Napište predikát `zip/3`, který ze dvou seznamů vytvoří nový spojením příslušných prvků do dvojic. Například dotaz `zip([1,2,3], [4,5,6], S).` uspěje se substitucí `S = [1-4,2-5,3-6]`. V případě rozdílných délek seznamů se přebytečné prvky ignorují.

[1] http://www.swi-prolog.org/pldoc/doc_for?object=section%282,%274.7%27,swi%28%27/doc/Manual/compare.html%27%29%29

Cvičení 12

12.1 Řezy

Příklad 12.1.1 Navrhněte predikát `max/3`, který uspěje, jestliže je číslo ve třetím argumentu maximem čísel z prvních dvou argumentů. Uveďte řešení bez použití řezu i s ním.

Příklad 12.1.2 Jaký je rozdíl mezi následujícími definicemi predikátů `member/2`? Ve kterých odpovědích se budou lišit?

a)
```
mem1(H, [H|_]).
mem1(H, [_|T]) :- mem1(H, T).
```

b)
```
mem2(H, [H|_]) :- !.
mem2(H, [_|T]) :- mem2(H, T).
```

c)
```
mem3(H, [K|_]) :- H == K.
mem3(H, [K|T]) :- H \== K, mem3(H, T).
```

Příklad 12.1.3 Zakreslete výpočetní SLD stromy následujícího programu na dotaz `?- b(X, Y).` Zakreslete také, kde se nacházejí větve, kterými výpočet nepokračuje, a popište, o které použití řezu se jedná (ořezání/upnutí).

a)
```
a(X) :- X = 0.
a(X) :- X = 1, !.
a(X) :- X = 2.
b(X, Y) :- a(X), a(Y).
```

b)
```
a(X) :- X = 0.
a(X) :- X = 1.
a(X) :- X = 2.
b(X, Y) :- a(X), !, a(Y).
b(X, Y) :- a(X), a(Y).
```

Příklad 12.1.4 Napište predikát `insert/3`, který do uspořádaného seznamu čísel vloží další číslo tak, aby i výslední seznam byl uspořádán. Například dotaz `?- insert(7, [1,2,3,10,12], X).` uspěje se substitucí `X = [1,2,3,7,10,12]`.

Příklad 12.1.5 Napište predikát `remove/3`, který odstraní všechny výskyty prvního argumentu ze seznamu ve druhém argumentu a výsledný seznam unifikuje do třetího argumentu.

Příklad 12.1.6 Napište predikát `intersection/3` pro výpočet průniku dvou seznamů a predikát `difference/3` pro výpočet rozdílu dvou seznamů. Můžete předpokládat, že žádný seznam neobsahuje stejný prvek vícekrát.

Příklad 12.1.7 Níže uvedený logický program obsahuje predikát `fib/2`, který počítá n-tý člen Fibonacciho posloupnosti ($n \geq 1$). Program však není plně korektní: správnou hodnotu sice vypočítá, ale když si pak pomocí `;` vyžádáme další řešení, program se zacyklí (měl by skončit

s odpovědí `no`). Přidejte do programu na správná místa operátory řezu tak, aby v tomto případu necyklil.

```
fib(N, 1) :- N =< 2.
fib(N, X) :- N1 is N - 2, fib(N1, 1, 1, X).
fib(0, _A, X, X).
fib(N, A, B, X) :- N1 is N - 1, C is A + B, fib(N1, B, C, X).
```

Příklad 12.1.8 Do programu z úlohy 10.3.3 přidejte jeden řez tak, aby výsledný program uspěl se stejným dotazem (`?- p(X, X).`) právě dvakrát.

12.2 Negace

Příklad 12.2.1 Definujte predikát `nd35/1`, který je pravdivý, pokud jako parametr dostane číslo, které není beze zbytku dělitelné čísly 3 ani 5. Úlohu vyřešte bez použití řezu a negace, s použitím řezu a s použitím negace.

12.3 Predikáty pro všechna řešení

Příklad 12.3.1 S využitím predikátů pro všechna řešení a knihovních funkcí pro seznamy napište následující predikáty:

a) `variation3all(+List, ?Variations)`, který vygeneruje seznam všech tříprvkových variací prvků ze seznamu `List`.
b) `combination3all(+List, ?Combinations)`, který vygeneruje seznam všech tříprvkových kombinací prvků ze seznamu `List`.

Poznámka: Podívejte se i na úlohu 11.2.6.

Příklad 12.3.2 Uvažme následující databázi faktů:

```
f(a, b).
f(a, c).
f(a, d).
f(e, c).
f(g, h).
f(g, b).
f(i, a).
```

Bez použití interpretru zjistěte, s jakou substitucí uspějí následující dotazy:

a) `?- findall(X, f(a, X), List).`
b) `?- findall(X, f(X, b), List).`
c) `?- findall(X, f(X, Y), List).`
d) `?- bagof(X, f(X, Y), List).`
e) `?- setof(X, Y ^ f(X, Y), List).`

Příklad 12.3.3 Napište predikát `subsets/2`, který pro danou množinu vygeneruje všechny její podmnožiny. Množinou rozumíme seznam, ve kterém se neopakují prvky. Na pořadí vygenerovaných množin nezáleží. Napište i predikát `isset/1`, který uspěje, když zadaný seznam korektně reprezentuje množinu (tedy neobsahuje duplicitní prvky).

12.4 Základy I/O

Příklad 12.4.1 Napište predikát `fileSum(+FileName, -Sum)`, který bude číst zadaný soubor po termech a spočítá sumu čísel v termech tvaru `s(X)`. Ostatní termy bude ignorovat.

Příklad 12.4.2 Napište predikát `contains(+FileName, +Char)`, který uspěje, pokud se v souboru se jménem `FileName` vyskytuje znak `Char`.

Příklad 12.4.3 S použitím predikátů pro získání všech řešení a predikátu `contains/2` z předchozího příkladu získejte všechny znaky uvedené v daném souboru.

Cvičení 13

13.1 Logické programování s omezujícími podmínkami

Příklad 13.1.1 Vyřešte následující logickou úlohu. Jednotlivá slova reprezentují čísla, každé písmeno zastupuje jednu číslici, různá písmena představují různé číslice.

$$SEND + MORE = MONEY$$

Řešení využívá logické programování s omezujícími podmínkami v konečných doménách, nezapomeňte proto do programu zahrnout také načítání správné knihovny pomocí následujícího řádku:

```
:- use_module(library(clpfd)).
```

Příklad 13.1.2 Vyřešte následující algebrogram. Jednotlivá slova reprezentují čísla, každé písmeno zastupuje jednu číslici, různá písmena představují různé číslice. Nalezený výsledek přehledně vypište na obrazovku.

$$DONALD + GERALD = ROBERT$$

Příklad 13.1.3 Vyřešte pomocí Prologu následující algebrogram:

$$\begin{array}{ccccc} \text{KC} & + & \text{I} & = & \text{OK} \\ + & & + & & + \\ \text{A} & + & \text{A} & = & \text{KM} \\ = & & = & & = \\ \text{OL} & + & \text{KO} & = & \text{LI} \end{array}$$

Příklad 13.1.4 Napište predikát `fact/2`, který počítá faktoriál pomocí CLP. V čem je lepší než klasicky implementovaný faktoriál?

Příklad 13.1.5 S využitím knihovny `clpfd` implementujte program, který spočítá, kolika a jakými mincemi lze vyskládat zadanou částku. Snažte se, aby řešení s menším počtem mincí byla preferována (nalezena dříve).

Příklad 13.1.6 Uvažte problém osmi dam. Cílem je umístit na šachovnici 8×8 osm dam tak, aby se žádné dvě neohrožovali. Dámy se ohrožují, pokud jsou ve stejném řádku, sloupci nebo na stejné diagonále.

S pomocí knihovny `clpfd` napište predikát `queens/3`, který dostane v prvním argumentu rozměr šachovnice, ve druhém výstupním argumentu bude jako výsledek seznam s pozicemi sloupců, do kterých třeba v jednotlivých řádcích umístit dámy, a ve třetím argumentu bude možné ovlivnit způsob hledání hodnot (predikát `labeling/2`).

Příklad výstupu:

```
?- queens(8, L, [up]).
L = [1,5,8,6,3,7,2,4]
...
```

Příklad 13.1.7 Napište program, který nalezne řešení zmenšené verze Sudoku. Hrací pole má rozměry 4×4 a je rozdělené na 4 čtverce 2×2. V každém řádku, sloupci a čtverci se musí každé z čísel 1 až 4 nacházet právě jednou. Jako rozšíření můžete přidat formátovaný výpis nalezeného řešení.

Příklad 13.1.8 Stáhněte si program *sudoku.pl*. Soubor naleznete v ISu a v příloze sbírky.

a) Program spusťte a naučte se jej ovládat. Zamyslete se, jak byste funkcionalitu programu sami implementovali.
b) Prohlédněte si zdrojový kód programu a pochopte, jak funguje.
c) Modifikujte program tak, aby nalezená řešení splňovala podmínku, že na všech políčkách hlavní diagonály se vyskytuje pouze jedna hodnota.

13.2 Opakování

Příklad 13.2.1 Určete, které dotazy uspějí:

a) `?- X = 1.`
b) `?- X == 1.`
c) `?- X =:= 1.`
d) `?- X is 1 + 1.`
e) `?- X = 1, X == 1.`
f) `?- X = 1 + 1, X == 2.`
g) `?- X = 1 + 1, X =:= 2.`
h) `?- g(X, z(X)) = g(Y, Z).`
i) `?- a(a, a) = X(a, a).`
j) `?- a(1, 2) = b(1, 2).`

Příklad 13.2.2 Opravte chyby v následujícím programu a vylepšete jeho nevhodné chování (bez použití CLP). Měl by fungovat správně pro celá čísla a můžete předpokládat, že první argument je vždy plně instanciován.

```
fact(N, Fact) :-
    M #= N - 1, fact(M, FactP), Fact #= N * FactP.
fact(0, 1).
```

Příklad 13.2.3 Napište predikát `merge/3`, který spojí 2 vzestupně uspořádané seznamy čísel z prvního a druhého argumentu. Výsledný vzestupně uspořádaný seznam pak unifikuje do třetího argumentu. Například dotaz `merge([1,2,4], [0,3,5], X)` uspěje se substitucí `X = [0,1,2,3,4,5]`.

Poté naprogramujte predikát `mergesort/2`, který seřadí seznam čísel z prvního argumentu podle velikosti s využitím techniky *merge sort* a výsledek unifikuje s druhým argumentem. Může se vám hodit i pomocný predikát `split/3`, který rozdělí zadaný seznam na 2 seznamy stejné délky.

Příklad 13.2.4 Napište predikát `quicksort/2`, který seřadí seznam v prvním argumentu metodou *quick sort* a výsledek unifikuje do druhého argumentu. Pravděpodobně se vám bude hodit i pomocný predikát `split/4`, který podle zadaného pivota rozdělí seznam na prvky menší než zadaný pivot a prvky větší nebo rovny než tento pivot.

Příklad 13.2.5 S využitím Prologu vyřešte Einsteinovu hádanku.

Zadání

- Je 5 domů, z nichž každý má jinou barvu.
- V každém domě žije jeden člověk, který pochází z jiného státu.
- Každý člověk pije jeden nápoj, kouří jeden druh cigaret a chová jedno zvíře.
- Žádní dva z nich nepijí stejný nápoj, nekouří stejný druh cigaret a nechovají stejné zvíře.

Nápovědy

1. Brit bydlí v červeném domě.
2. Švéd chová psa.
3. Dán pije čaj.
4. Zelený dům stojí hned nalevo od bílého.
5. Majitel zeleného domu pije kávu.
6. Ten, kdo kouří *PallMall*, chová ptáka.
7. Majitel žlutého domu kouří *Dunhill*.
8. Ten, kdo bydlí uprostřed řady domů, pije mléko.
9. Nor bydlí v prvním domě.
10. Ten, kdo kouří *Blend*, bydlí vedle toho, kdo chová kočku.
11. Ten, kdo chová koně, bydlí vedle toho, kdo kouří *Dunhill*.
12. Ten, kdo kouří *BlueMaster*, pije pivo.
13. Němec kouří *Prince*.
14. Nor bydlí vedle modrého domu.
15. Ten, kdo kouří *Blend*, má souseda, který pije vodu.

Úkol Zjistěte, kdo chová rybičky.

Postup

a) Uvažme řešení uvedené v souboru *einstein.pl*. Soubor naleznete v ISu a v příloze sbírky.
b) Pochopte, jak by měl program pracovat. Vysvětlete, proč na dotaz `?- rybicky(X).` program zdánlivě cyklí.
c) Přeuspořádejte pravidla tak, aby Prolog našel řešení na tento dotaz do jedné vteřiny.
d) Odstraňte kategorizaci objektů (`barva/1`, `narod/1`, `zver/1`, `piti/1`, `kouri/1`) a požadavky na různost entit v každé kategorii. Diskutujte, v jaké situaci, by vám tato kategorizace byla prospěšná.
e) Nyní sdružte entity stejného typu do seznamů. Modifikujte program tak, aby místo predikátu `reseni/25` používal predikát `reseni/5`.
f) Podobnou modifikaci proveďte i s jednotlivými pravidly, tj. místo `ruleX/10` použijte `ruleX/2` a místo `ruleX/5` použijte `ruleX/1`.
g) Definujte pomocné predikáty `isLeftTo/4`, `isNextTo/4`, `isTogetherWith/4`, které prověřují požadované vztahy, například: `isNextTo(A, B, Acka, Bcka)` je pravdivý, pokud se objekt `A` vyskytuje v seznamu `Acka` na pozici, která sousedí s pozicí objektu `B` v seznamu `Bcka`. Tímhle způsobem přepište všechna pravidla.
h) Obdivujte krásu výsledku po vašich úpravách.

Jiná kódování Přeformulujte program tak, aby predikát `reseni/5` jako své argumenty bral seznamy objektů odpovídající jednotlivým pozicím v ulici, tj. 5 seznamů takových, že každý seznam bude obsahovat informaci o barvě domu, národnosti obyvatele, chovaném zvířeti, oblíbeném kuřivu a oblíbeném nápoji obyvatele.

Příklad 13.2.6 Napište predikát `equals/2`, který uspěje, pokud se dva zadané seznamy rovnají. V opačném případě vypíše důvod, proč je tomu tak (jeden seznam je kratší, výpis prvků, které se nerovnají, …). Můžete předpokládat, že seznamy neobsahují proměnné. Příklady použití najdete níže.

```
?- equals([5,3,7], [5,3,7]).
true.
?- equals([5,3,7], [5,3,7,2]).
1st list is shorter
false.
?- equals([5,3,7], [5,2,3,7]).
3 does not equal 2
false.
```

Příklad 13.2.7 Nechť je zadána databáze faktů ve tvaru `road(a, b)`, které vyjadřují, že z místa `a` existuje přímá (jednosměrná) cesta do místa `b`. Napište predikát `trip/2` tak, že dotaz `?- trip(a, b).` uspěje, jestli existuje nějaká cesta z místa `a` do místa `b`. Můžete předpokládat, že cesty netvoří cykly (tedy pokud jednou z nějakého místa odejdete, už se tam nedá vrátit).

Rozšíření 1
Upravte vaše řešení tak, že predikát `trip(a, b, S)` uspěje, jestli existuje nějaká cesta z místa `a` do `b` a `S` je seznam měst, přes které tato cesta prochází (v daném pořadí).

Rozšíření 2
Upravte vaše řešení tak, aby fungovalo i v případě, že cesty mohou tvořit cykly.

Řešení

Řešení 1.1.1

a) V prvním výrazu je implicitní závorkování v důsledku priorit operátorů kolem násobení, tj. `5 + (9 * 3)`.
b) Operace umocňování má asociativitu 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)`

   Ale vidíme, že tato rovnost platí pro `n == 2`, tedy to je jediný speciální případ, kdy `n ^ n ^ n == (n ^ n) ^ n`.
c) Tady se setkáváme s případem, kdy je operátor neasociativní, tedy není definováno, jak se výraz parsuje v případě výskytu více relačních operátorů vedle sebe, a je tedy nekorektní. Důvod neasociativity je jednoduchý. Asociativitu má smysl uvažovat jen u operátorů, které mají stejný typ obou operandů a také výsledku. To zřejmě neplatí u operátoru `(==)`, který vyžaduje operandy stejného, ale jinak poměrně libovolného typu, například čísla, Boolovské hodnoty, řetězce, ..., ale výsledek je Boolovská hodnota. Pokud bychom tedy nějak asociativitu definovali, dospěli bychom v některých případech do situace, kdybychom porovnávali Boolovskou hodnotu s hodnotami jiného typu, což v Haskellu nelze.
d) v případě, že explicitně uvedeme závorkování pro relační operátory, dostaneme se do obdobné situace jako v předchozím podpříkladu, tedy porovnání Boolovské hodnoty a čísla, což v Haskellu nelze. Naproti tomu výsledkem porovnání `4 == 4` dostaneme v obou případech Boolovskou hodnotu, a ty mezi sebou porovnávat můžeme, protože jsou stejného typu.

Řešení 1.1.2 V uvedeném pořadí od nejvyšší priority (9) až k nejnižší (1).

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

Řešení 1.1.3

a) Podmínkou musí být výraz Boolovského typu (`Bool`), což výraz `5 - 4` není -- jde o výraz celočíselného typu.
b) Výrazy v `then` a `else` větvi musí být stejného (nebo kompatibilního) typu, protože celý podmínkový výraz musí mít vždy stejný typ bez ohledu na hodnotu podmínky.
c) Na první pohled podivně vypadající konstrukce, kde výsledkem podmínkového výrazu je prefixově zapsaný operátor `(&&)`, je správná. V Haskellu jsou funkce/operátory výrazy rovnocennými 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 jenom 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.

Řešení 1.1.4 Je důležité zachovávat pořadí operandů! I když jde o komutativní operátor, nelze při změně mezi prefixovým a infixovým zápisem měnit jejich pořadí, protože vzniklý výraz nebude ekvivalentní.

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

Řešení 1.1.5 Postup implicitního závorkování je u všech výrazů stejný. Je potřeba řídit se 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 (samozřejmě ignorujíce obsah explicitních závorek) a jejich operandy uzávorkujeme. Pokud je těchto operátorů více než jeden, jednotlivé operandy závorkujeme dle jejich asociativity.
2. Nejsou-li ve výrazu infixově zapsané operátory, ale jenom prefixové aplikace funkcí na argumenty, závorkujeme funkci s její argumenty zleva (ve smyslu „částečné aplikace“, bude bráno později), konkrétně například `f 1 2 'd' m` závorkujeme `(((f 1) 2) 'd') m`.

Pokud nejde realizovat ani jeden z těchto kroků, tj. výraz je jednoduchý (konstanta, proměnná nebo název funkce), aplikace funkce na jeden jednoduchý argument nebo aplikace infixově zapsaného binárního operátoru na dva jednoduché argumenty, skončili jsme.

Pokud ne a vzniklé uzávorkované podvýrazy jsou složitější, opět aplikujeme tento postup na všechny rekurzivně.

Řešení jednotlivých příkladů budou následovná:

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

Řešení 1.1.6 Nejdříve za pomoci tabulky 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)`

Řešení 1.1.7

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

Řešení 1.1.8

a) Korektní, „přičítačka“ tří.
b) Nekorektní, `3` je interpretována jako funkce beroucí `(+)` jako svůj argument.
c) Korektní, „přičítačka“ tří, ekvivalentní funkci `f x = 3 + x`.
d) Korektní, „přičítačka“ tří, ekvivalentní funkci `f x = x + 3`.
e) Korektní, ekvivalentní funkci `f x y = x ((+) y)`, například `f id 3 2` $\rightsquigarrow^*$ `5`.
f) Nekorektní, ekvivalentní s funkcí `f x = x + (.)`, avšak funkce není možné sečítat, typová chyba.
g) Nekorektní, není možné rozlišit, kterého operátoru to je operátorová sekce. Interpretuje se jako prefixová verze operátoru `(.+)`.
h) Nekorektní, viz předešlý příklad.
i) Nekorektní, zřejmě zamýšlená operátorová sekce `(.)`, avšak závorky okolo operátorové sekce není možné vynechat.
j) Korektní, dvojnásobné použití operátorové sekce -- vnější je pravá, vnitřní je levá. Ekvivalentní se všemi následujícími funkcemi.

```
f1 x = x . ((.).)
f2 x y = x (((.).) y)
f3 x y = x ((.) . y)
f4 x y = x (\z -> (.) (y z))
f5 x y = x (\z w q -> y z (w q))
```

Řešení 1.2.1

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
```

Řešení 1.2.2

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

Funkce je definována po částech a předpokládáme, že dostane jako argument jenom nezáporné celé číslo. Jako první bázový případ je 0, kdy přímo vrátíme výsledek. V opačném případě použijeme druhý rekurzivní případ, kdy víme, že $n! = n \times (n-1)!$. Poznamenejme, že závorky kolem `n - 1` je nutno použít, protože jinak by se výraz implicitně uzávorkoval jako `(fact n) - 1`, protože aplikace funkcí na argumenty má vyšší prioritu než operátory.

Řešení 1.2.3 Nejjednodušším, i když ne nejefektivnějším způsobem úpravy je doplnit kontrolu nezápornosti argumentu v druhém případě definice funkce, tedy:

```
z `power` n = if n < 0 then error "negative!" else z * (z `power` (n-1))
```

Neefektivita spočívá v tom, že kontrolu nezápornosti stačí udělat pouze jednou na začátku, protože pak už nelze dosáhnout záporného čísla. V našem případě se kontrola provede zbytečně n-krát.

Řešení 1.2.4

```
dfct :: Integer -> Integer
dfct 0 = 1
dfct 1 = 1
dfct n = n * dfct (n - 2)
```

Řešení 1.2.5 Musíme rozlišit následující tři případy:

a) rovnice má nekonečně mnoho řešení ($a = b = 0$)
b) rovnice nemá řešení ($a = 0 \wedge b \neq 0$)
c) rovnice má právě jedno řešení tvaru $\frac{-b}{a}$ ($a \neq 0$)

Funkci zadefinujeme podle vzoru následovně (pozor na pořadí vzorů!).

```
linear :: Double -> Double -> Double
linear 0 0 = error "Infinitely many solutions"
linear 0 _ = error "No solution"
linear a b = -b/a
```

Řešení 1.2.6 Využijeme funkci `fact` definovanou na cvičeních. Jelikož pracujeme s celými čísly, musíme použít celočíselné dělení místo reálného (jinak bychom porušili deklarovaný typ).

```
combinatorial :: Integer -> Integer -> Integer
combinatorial n k = div (fact n) ((fact k) * (fact (n-k)))
```

Řešení 1.2.7 Jak víme, počet kořenů kvadratické rovnice závisí na hodnotě diskriminantu -- pro záporný diskriminant rovnice řešení nemá, pro nulový má právě jedno a pro kladný právě dvě. Funkci můžeme definovat pomocí podmíněného výrazu a lokální definice například následovně:

```
roots :: Double -> Double -> Double -> Int
roots a b c = if d < 0 then 0
                       else if d == 0 then 1 else 2
    where d = b ^ 2 - 4 * a * c
```

S využitím funkce `signum` můžeme naše řešení značně zkrátit -- zamyslete se, proč je to možné. (Poznámka: funkce `signum` vrátí výsledek stejného typu, jako je její argument, musíme tedy použít některou ze zaokrouhlovacích funkcí pro převod `Double` na `Int`.)

```
roots' :: Double -> Double -> Double -> Int
roots' a b c = floor (1 + signum (b ^ 2 - 4 * a * c))
```

Řešení 1.2.8

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

Řešení 1.2.9 K řešení použijeme známý Euklidův algoritmus, konkrétněji jeho rekurzivní verzi využívající zbytky po dělení.

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

Řešení 1.2.10

```
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))
```

Řešení 1.2.11 Tuto úlohu je možno řešit různými způsoby. Asi nejpřímočařejší je porovnávat hodnotu se všemi mocninami dvou, které ji nepřesahují. Je tedy nutné použít ještě pomocnou binární funkci, která bude mít aktuálně zpracovávanou mocninu jako svůj druhý argument.

```
power2 :: Integer -> Bool
power2 x = power2' x 1
power2' :: Integer -> Integer -> Bool
power2' x y = if y > x then False
                       else if x == y then True else power2' x (y * 2)
```

Existují však i mnohem efektivnější řešení. Jedno z nich, postavené na logaritmech, je uvedeno níže; musíme však vzít v úvahu, že funkce `log` očekává jako svůj parametr desetinné číslo, proto musíme `x` konvertovat pomocí funkce `fromIntegral`:

```
power2 :: Integer -> Bool
power2 x = 2 ^ (floor (log (fromIntegral x) / log 2)) == x
```

Řešení 1.2.12 Nejsnáze to lze zjistit výpočtem několika prvních hodnot. Pak lze snadno odhalit zákonitost a formulovat hypotézu, že `fun n == n ^ 2` pro nezáporná `n`. Následně je potřeba tuto hypotézu dokázat, což lze provést matematickou indukcí. Důkaz přenecháváme čtenáři.

Na tuto definici lze taky nahlížet tak, že všechny druhé mocniny se dají rozepsat jako součet lichých čísel nepřevyšujících dvojnásobek argumentu.

V případě záporných čísel lze ručním vyhodnocením zjistit, že vyhodnocování se „zacyklí" na druhém řádku definice a bude postupně voláno pro nižší a nižší záporná čísla. Definice totiž neposkytuje žádný zastavovací případ, který by rekurzi ukončil.

Řešení 1.3.1

a) `let t = (3+4)/2 in t * (t - 3) * (t - 4)`
b) Soubor bude obsahovat řádek `t = (3 + 4) / 2` a v interpretru pak lze po načtení tohoto souboru zadat k vyhodnocení výraz `t * (t - 3) * (t - 4)`.

Řešení 1.3.2 Jsou dvě varianty -- první funguje jen pro nezáporná čísla (zkuste si ji rozšířit), druhá je pravděpodobně efektivnější (přičemž to záleží i na implementaci `read`).

```
flipNum :: Integer -> Integer
flipNum x = read (reverse (show x))

flipNum' :: Integer -> Integer
flipNum' x = fromDigits 0 (reverse (toDigits x []))
  where
    toDigits 0 xs = xs
    toDigits n xs = toDigits (n `div` 10) (n `mod` 10 : xs)
    fromDigits n []     = n
    fromDigits n (x:xs) = fromDigits (n * 10 + x) xs
```

Na druhou stranu je první verze podstatně elegantnější, což se cení. Navíc ji lze ještě zlepšit na `flipNum = read . reverse . show`.

Řešení 2.1.1 Použijte příkazu `:t` k otypování výrazu v `ghci` (typ `[Char]` je ekvivalentní typu `String`).

Řešení 2.1.2

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 interpretru 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) `()`, takzvaná nultice je typem s jedinou hodnotou (někdy také označujeme jako jednotkový typ, v angličtině *unit*, v podstatě odpovídá typu `void` v C). 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.
f) `(1, 1)`, `(42, 16)`, `(10 - 5, 10 ^ 10000)`, …
Dvojice, první výraz musí být typu `Int`, druhý typu `Integer`.

g) `(0, 3.14, True)`, ...Trojice, složky musí odpovídat typům.
h) `((), (), ())` je jediná možná hodnota trojice jejímž každým prvkem je nultice.

Řešení 2.1.3

a) `Bool`, výraz je datovým 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: `(&&) :: Bool -> Bool -> Bool`, `True :: Bool`, `"1" :: String`. 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) `Double`, výraz `3.14` může být typu `Double`.

Řešení 2.1.4

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

Řešení 2.1.5 Vidíme, že `f` i `x` jsou funkce. Předpokládejme zatím tedy, že `x :: a1 -> a2` a `f :: b1 -> b2`. Z aplikací ve výrazu vyplývá, že `b1 = a1 -> a2` a `b2 = a1`.

Typ funkce `f` tedy musí být unifikovatelný s typem `(a -> b) -> a`.

Řešení 2.1.6

a) Ne, `id :: x -> x`, vznikne problém `a = b -> a` (nekonečný typ).
b) Ne, `const :: x -> y -> x`, problémy `a = y -> a -> b` a `(y -> x) -> b = x` (nekonečné typy).
c) Ano, `const :: x -> y -> x, (a -> b) -> y -> a -> b`.
d) Ne, `map :: (x -> y) -> [x] -> [y]`, vznikne problém `[y] = c -> d -> e` (nekompatibilní typy).

Existuje jednoduchý způsob, jak tuto úlohu řešit v interpretru -- ten umožňuje unifikaci libovolného konečného počtu funkcí a typů. Stačí zadat

```
:t [undefined :: t1, ..., undefined :: tm, f1, ..., fn]
```

kde `t1` až `tm` jsou typy a `f1` až `fn` jsou funkce.

Řešení 2.2.1 Funkce `min` vrací menší z dvou argumentů. Tedy máme dva případy. Když druhý argument bude menší než 6, výsledkem funkce bude tento argument. V opačném případě, když druhý argument bude alespoň 6, výsledkem bude 6 jako to menší z dvojice čísel.

```
min6 :: (Num a, Ord a) => a -> a
min6 x = if x < 6 then x else 6
```

Typ funkce `min6` je poněkud složitější. Jeho význam není teď důležitý a bude vysvětlen později.

Řešení 2.2.2

a) Ne.
b) Ano, `(f 1 g) 2` ≡ `f 1 g 2` ≡ `(f 1) g 2`.
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 operátora `(+)` nevynucuje). Nezapomínejme, že všechny operátory můžeme předefinovat. *Poznámka*: (pokročilejší) Toto by bylo možné pouze v případu, že by komutativity vyžadovali axiomy typové třídy, ve které je daný operátor/funkce definována. Ani to by však nezaručovalo skutečnou korektnost -- interpret/kompilátor platnost axiom nekontroluje (ani to není v jeho silách). Zůstává pouze důvěra v programátora, že jeho implementace je korektní.
d) Ne, opět není možné přehodit parametry operátoru `+`.
e) Ne, je nutné uzávorkovat druhý argument.
   `81 * f 2` ≡ `81 * (f 2)` ≡ `(*) 81 (f 2)`
f) Ne, je nutné přidat závorku k argumentu na konci.
   `fact n` ⇝ `n * fact (n - 1)`
g) Ano (použít závorky v tomto případě není nutné).
h) Ne, protože `.` ve výrazu `sin 1 . 43` je operátor (skládání funkcí), zatímco ve výrazu `sin 1.43` se jedná o desetinnou tečku.
i) Ne, `8 - 7 * 4` ≡ `(-) 8 ((*) 7 4)`.

Řešení 2.3.1

a) OK, typ `[Integer]`
b) chybné, `(1:2)` je chybný výraz, protože 2 není seznam
c) OK, ekvivalentní a
d) OK, ekvivalentní a, c
e) chybné, různé typy prvků: `1 :: Integer` ale `'a' :: Char`
f) OK, typ `[[Integer]]`
g) chybné, různé typy prvků: `1 :: Integer` ale `[1,2] :: [Integer]`
h) OK, typ `[[a]]`

Řešení 2.3.2 Použijte příkazu `: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]`, třeba si dát pozor při otypování takovýchto výrazů. Výraz sice obsahuje funkci `++`, která má v tomto kontextu typ `String -> String -> String`, avšak `String -> String -> String` není výsledný typ, protože na funkci už byli aplikovány 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) `[[[Char]]]` (což je stejné jako `[[String]]`), typové omezení vzniká kvůli konkrétní hodnotě ve druhém prvku (prázdný řetězec).

Řešení 2.3.3

- `[]`
  Tento vzor představuje prázdný seznam. Nemůže reprezentovat žádný z uvedených seznamů.
- `x`
  Libovolná hodnota (a tedy libovolný seznam) se může navázat na tento vzor. Může reprezentovat všechny uvedené seznamy.
- `[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.
- `(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, kterého 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, kterého 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]]`.

Řešení 2.3.4

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

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

Řešení 2.3.5 Funkci definujeme po částech. Případ prázdného seznamu nemusíme řešit. Dalším větším seznamem je jednoprvkový seznam a v tomto případě vrátíme rovnou jeho jediný prvek:

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

Všechny zbývající případy seznamů mají dva nebo více prvků. Hledaný poslední prvek u nich získáme tak, že budeme postupně odstraňovat prvky ze začátku seznamu. Tedy ze zadaného prvku odstraníme první prvek a na zbytek aplikujeme opět funkci `getLast`:

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

Řešení 2.3.6 Funkci definujeme po částech, 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 případu bude první prvek zadaného seznamu určitě ve výsledném seznamu a o zbytku seznamu musíme rozhodnout rekurzivně:

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

Srovnejte s definicí funkce `getLast`.

Řešení 2.3.7

```
median :: [a] -> a
median [x] = x
median [x, _] = x
median (_:s) = median (init s)
```

Řešení 2.3.8

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

Výpočet této funkce probíhá například takto:

`len (1:(2:[]))` $\leadsto$ `1 + len (2:[])` $\leadsto$ `1 + (1 + len [])` $\leadsto$ `1 + (1 + 0)` $\leadsto^*$ `2`

Řešení 2.3.9

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

Řešení 2.3.10

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

Řešení 2.3.11

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

Příklad výpočtu:

```
multiplyN 2 (1:(2:[])) ⇝ 1 * 2 : multiplyN 2 (2:[])
    ⇝ 1 * 2 : (2 * 2 : multiplyN 2 []) ⇝ 1 * 2 : (2 * 2 : [])
    ⇝* 2 : (4 : []) ≡ [2, 4]
```

Řešení 2.3.12 Inspirujeme se funkcí `multiplyN` a zobecníme ji na libovolnou funkci. Tím dostaneme tento předpis:

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

Řešení 2.3.13

```
add1        :: [Integer] -> [Integer]
add1        = applyToList (+1)
multiplyN   :: Integer -> [Integer] -> [Integer]
multiplyN n = applyToList (*n)
```

Řešení 2.3.14

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

Řešení 2.3.15

```
evens :: [Integer] -> [Integer]
evens = filter even
```

Řešení 2.3.16

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

Řešení 2.3.17

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

multiplyEven' :: [Integer] -> [Integer]
multiplyEven' = multiplyN 2 . filter even
```

Fungovalo by složení funkcí v opačném pořadí? Jakým číslem bychom museli násobit?

Řešení 2.3.18

```
sqroots :: [Double] -> [Double]
sqroots = map sqrt . filter (>0)
```

Řešení 2.3.20 Jedním z možných řešení je použití funkce `reverse` a operátora `!!` na výběr prvku podle pozice v seznamu (indexuje se od nuly).

```
fromend :: Int -> [a] -> a
fromend x s = (reverse s) !! (x-1)
```

Další možností je využití funkce length -- jestli je délka seznamu menší než zadaný argument, funkce skončí s chybovou hláškou, jinak vybereme prvek podle jeho indexu.

```
fromend' :: Int -> [a] -> a
fromend' x s = if x > len then error "Too short"
                          else s !! (len - x)
    where len = length s
```

Zkuste se zamyslet, které z uvedených řešení bude mít menší časovou složitost. Je možné napsat i rychlejší funkci?

Řešení 2.3.21 Využívajíc knihovní funkce `map` je řešení velmi krátké.

```
maxima :: [[Int]] -> [Int]
maxima s = map maximum s
```

Řešení 2.3.22 Nejdřív si zadefinujeme pomocný predikát `isvowel`, který o znaku určí, jestli je samohláskou. Následně jednotlivé řetězce projdeme knihovní funkcí `filter`.

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

Řešení 2.3.23 Funkci, která rozhodne, jestli je řetězec palindromem, zadefinujeme jednoduše pomocí funkce `reverse` a porovnání.

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

Po krátkém zamyslení zjistíme, že na doplnění slova na palindrom nám stačí najít část slova, která tvoří palindrom, a vznikne vynecháním několika prvních písmen. Vynechané znaky pak doplníme 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.

Řešení 2.3.24

```
brackets :: String -> Bool
brackets s = br s 0 where
  br [] k = k == 0
  br (x:xs) k = if x == '('
    then br xs (k + 1)
    else if k <= 0 then False else br xs (k - 1)
```

Řešení 2.3.25

```
domino :: (Eq a) => [(a,a)] -> [(a,a)]
domino [] = []
domino ((a,b):xs) = (a,b) : domino' b xs
domino' :: (Eq a) => a -> [(a,a)] -> [(a,a)]
domino' k s = domino (dropWhile ((/=k) . fst) s)
```

Řešení 2.3.26

```
s2m n = map (^2) [0..n]
```

Řešení 3.1.1

a) Intuitivně se výraz vyhodnocuje tak, že postupně aplikujeme skládané funkce odzadu, výsledek je tedy `True`. Po krocích můžeme výraz vyhodnotit takto (s ohledem na definici `(.) f g x = f (g x)`):

`((== 42) . (2 +)) 40` ⇝ `(== 42) ((2 +) 40)` ⇝ `(== 42) 42` ⇝ `True`

b) `((> 2) . (* 3) . ((-) 4)) 5`
   ≡ `((> 2) . ( (* 3) . ((-) 4) )) 5`
   ⇝ `(> 2) ( ((* 3) . ((-) 4) 5 )`
   ⇝ `(> 2) ((* 3) ((-) 4 5))` ⇝ `(> 2) ((* 3) (-1))`
   ⇝ `(> 2) (-3)` ⇝ `False`

c) Filtrujeme seznam funkcí `((>= 2) . fst)`, která očekává dvojice a rozhoduje, zda je první složka této dvojice větší než 2 (první složka tedy musí být číslo, druhá může být cokoli).
Aplikací této funkce na náš seznam tedy dostaneme seznam těch hodnot, které mají první složku větší nebo rovnou 2, tedy

`filter ((>= 2) . fst) [(1, "a"), (2, "b"), (3, "c")]`
   ⇝* `[(2, "b"), (3, "c")]`

Řešení 3.1.2

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 3.1.3

a) `f . (g x)`
b) `(((f (.)) g) (h x)) . ((((.) f) g) x)`

Řešení 3.2.1

a) Typy základních podvýrazů jsou `(&&) :: Bool -> Bool -> Bool` a `True :: Bool`. Typ reálného prvního argumentu funkce `(&&)` souhlasí s typem prvního argumentu v typové deklaraci funkce, tedy `(&&)` lze aplikovat na parametr `True`, čímž se tento parametr naváže a výsledná funkce je typu `(&&) True :: Bool -> Bool`.
b) `id :: a -> a`, `"foo" :: String`. Typ prvního reálného parametru je konkrétnější, než typ formálních parametrů v deklaraci, tedy substituujeme `a` $\sim$ `String`. Po aplikaci na jediný parametr nám vychází typ `id "foo" :: String`.
c) `(&& False)` je pravá operátorová sekce operátoru `(&&) :: Bool -> Bool -> Bool`. Typ reálného parametru souhlasí s typem formálního parametru, tedy můžeme aplikovat. Pozor, aplikujeme však druhý parametr. Výsledný typ je tedy `(&& False) :: Bool -> Bool`.
d) `const :: a -> b -> a`, `True :: Bool`. Reálný parametr má konkrétnější typ -- substituce `a` $\sim$ `Bool`, tedy celkový typ je `const True :: b -> Bool`.
e) Obdobně jako v předchozím případě, jen navíc aplikujeme na `False`, tedy substituce `b` $\sim$ `Bool`. Výsledek je `const True False :: Bool`.
f) `(:[])` je pravá operátorová sekce operátoru `(:) :: a -> [a] -> [a]`. Typ reálného argumentu `[] :: [a]` souhlasí s typem formálního argumentu, můžeme tedy aplikovat (opět druhý argument). Výsledný typ je tedy `(:[]) :: a -> [a]`.
g) `(:[]) :: a -> [a]`, `True :: Bool`. Typ reálného argumentu je konkrétnější, substituujeme `a` $\sim$ `Bool`, výsledný typ je `(:[]) True :: [Bool]`.
h) `(:) :: a -> [a] -> [a]` sdružuje zprava, tedy výraz odpovídá seznamu `[ [], [] ]`. Jeho oba prvky jsou typu `[] :: [a]`, a tedy seznam je homogenní, a tedy otypovatelný. Výsledný typ je `[]:[]:[] :: [[a]]`.
i) Můžeme zapsat jako seznam `[[[]]]`, což odpovídá typu `[[[]]] :: [[[a]]]`.

Řešení 3.2.2

a) Podvýrazy jsou typů `map :: (a -> b) -> [a] -> [b]` a `fst :: (c, d) -> c` (je nutné volit různé typové proměnné v různých podvýrazech, abychom nedostali do výpočtu závislosti, které tam nemají být).

Nyní musíme unifikovat typ prvního parametru v typu funkce `map`, tedy `(a -> b)` s typem skutečného prvního parametru: `a -> b` $\sim$ `(c, d) -> c`. Jedná se o funkční typ, tedy unifikujeme první parametr levé strany s prvním parametrem na pravé straně a tak dále. Tím dostáváme substituci `a` $\sim$ `(c, d)` a `b` $\sim$ `c` a budeme dosazovat pravou stranu do levé, protože pravá strana je specifičtější typ.
Typ funkce `map` v tomto výrazu je tedy `map :: ((c, d) -> c) -> [(c, d)] -> [c]`.
Nyní již můžeme funkci `map` dosadit první parametr a dostat typ celého výrazu: `map fst :: [(c, d)] -> [c]`, tedy naše funkce bere seznam dvojic a vrací seznam obsahující první složky těchto dvojic.

b) Typy podvýrazů jsou: `map :: (a -> b) -> [a] -> [b]`, `filter :: (c -> Bool) -> [c] -> [c]` a `not :: Bool -> Bool`.
Nejprve musíme otypovat podvýraz `filter not` a podle jeho typu potom určit typ celého výrazu. Unifikujeme tedy typ prvního parametru v definici `filter` s reálným typem prvního parametru: `c -> Bool` $\sim$ `Bool -> Bool`, a tedy `c` $\sim$ `Bool`. Po dosazení za `c` tedy dostaneme typ aplikace `filter not :: [Bool] -> [Bool]`.
Nyní unifikujeme typ prvního parametru funkce `map` s typem `filter not`: `a -> b` $\sim$ `[Bool] -> [Bool]`, a tedy `a` $\sim$ `[Bool]`, `b` $\sim$ `[Bool]`.
Dosazením do typu funkce `map` a aplikací dostáváme `map (filter not) :: [[Bool]] -> [[Bool]]`.

c) `const :: a -> b -> a`, `id :: c -> c` (nutno zvolit různé typové proměnné v různých výrazech), `'!' :: Char`, `True :: Bool`. Argumenty dosazujeme postupně a substituujeme:
   - pro `const id`: substituce `a` $\sim$ `c -> c`, dosazujeme konkrétnější do obecnějšího a dostáváme typ aplikace `const id :: b -> c -> c`
   - dále aplikujeme `const id` na `'!'`, substituce `b` $\sim$ `Char`, výsledek `const id '!' :: c -> c`
   - aplikujeme `const id '!'` na `True`, substituce `c` $\sim$ `Bool`, konečný výsledek `const id '!' True :: Bool`.

d) V tomto případě použijeme alternativní pohled na typování, kdy si výraz zkusíme vyhodnotit, a podle toho určit jeho typ. Nejprve připomeneme, jak je tento výraz implicitně uzávorkovaný: `((fst (fst, snd)) (snd, fst)) (True, False)` a nyní vyhodnocujeme:

```
((fst (fst, snd)) (snd, fst)) (True, False)
    ⇝ (fst (snd, fst)) (True, False)
    ⇝ snd (True, False)
    ⇝ False
```

A tedy nám vychází typ `((fst (fst, snd)) (snd, fst)) (True, False) :: Bool`.
Zde je však třeba být opatrný -- pokud by výsledný typ mohl být polymorfní, je jistější udělat si všechny typové substituce.

e) Opět nejprve zkusíme výraz vyhodnotit:

```
head [head] [tail] [[]]
    ⇝ head [tail] [[]]
    ⇝ tail [[]]
    ⇝ []
```

Zdálo by se tedy, že výsledek je typu `[] :: [t]`. To však není pravda, protože typ výsledku je ovlivněn všemi substitucemi, které nastaly při typování výrazu. Proto musíme výraz s polymorfním návratovým typem skutečně otypovat:

`head :: [a] -> a`, `[head] :: [[b] -> b]`, `[tail] :: [[c] -> [c]]`, `[[]] :: [[d]]`. V podvýrazu `head [head]` unifikujeme `[a]` ∼ `[[b] -> b]`, a tedy `a` ∼ `[b] -> b`, tudíž `head [head] :: [b] -> b`, a tedy jej lze dále aplikovat na `[tail] :: [[c] -> [c]]` se substitucí `[b]` ∼ `[[c] -> [c]]`, a tedy `b` ∼ `[c] -> [c]`. Dostáváme `head [head] [tail] :: [c] -> [c]`.
Tento výraz je nyní aplikován na výraz `[[]] :: [[d]]`, což znamená unifikaci `[c]` ∼ `[[d]]`, a tedy `c` ∼ `[d]`. Správný výsledný typ je tedy `head [head] [tail] [[]] :: [[d]]`, což je typ seznamu seznamů, a tedy různý od `[t]`, který jsme odhadly dříve (a je moc obecný).

Řešení 3.2.3

a) Funkci lze jednoduše intuitivně otypovat, protože vidíme, že ve výsledku jenom prohodí argumenty uspořádané dvojice. Tedy vstupní typ `(a, b)` převede na typ `(b, a)`. Platí tedy `swap :: (a, b) -> (b, a)`.

b) Opět otypujeme intuitivně. Víme, že `tail :: [a] -> [a]` a `head :: [a] -> a`. Pozor! Normálně je takovéto východiskové otypování cestou k záhubě. Při otypovávání funkcí/-výrazů/proměnných je vždy potřeba použít nové, čerstvé typové proměnné. Jinak zavedeme nežádoucí a nepravdivou rovnost mezi typy, která způsobí, že určený výsledný typ výrazu nebude správný (nebude dostatečně obecný, případně nebude možné výraz vůbec otypovat). V tomto případě si to můžeme dovolit, protože tam rovnost je (vstup `head` je výstupem `tail`). Argument, který vstoupí do funkce `cadr`, je tedy typu `[a]`, protože to vyžaduje funkce `tail`. Z ní dostaneme hodnotu opět typu `[a]`, a ten dáme jako argument funkci `head`, načež dostaneme hodnotu typu `a`. Tedy ve výsledku máme typ `[a] -> a`.

c) Víme, že typ funkce `head` je `[a] -> a`, což v dvojnásobné aplikaci znamená, že vstup musí být typu `[[a]]` a výstup typu `a`. Výsledkem je tedy `[[a]] -> a`.

d) V tomto případě vidíme, že `twice` vytvoří dvojitou aplikaci zadané funkce. Tento případ možná vypadá stejně jako předchozí, avšak je v něm významný rozdíl v tom, že zatímco dva výskyty funkce `head` byly zcela nezávislé, a tedy mohli mít odlišně specializovaný typ, `f` je fixována formálním argumentem funkce `twice` a musí mít v obou výskytech stejný typ. Avšak vidíme, že typ vstupu musí být stejný jako typ výstupu, a tedy `f :: a -> a`. Ve výsledku tedy máme `twice :: (a -> a) -> a -> a`.

e) Vidíme, že `g` a `h` jsou funkce, tedy nechť `g :: a -> b, h :: c -> d -> e`. Na základě shody typů díky aplikaci funkce na argumenty také vidíme, že `x :: c, y :: d, a` ∼ `e`. Další typová omezení už nejsou. Funkční typ, který budeme hledat, sestává z typů `g, h, x, y` a typu těla definice `comp12`. Ve výsledku tedy dostaneme `(a -> b) -> (c -> d -> a) -> c -> d -> b`. `comp12 :: (a -> b) -> (c -> d -> a) -> c -> d -> b`.

Řešení 3.2.4

a) 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 :: [t] -> String`.

b) Z použitých vzorů můžeme odvodit, že funkce bere dva argumenty, první typu `Bool` a druhý je dvojice. Uvažujme tedy, že bude mít typ `(a, b)`.

Z toho tedy usoudíme na typy argumentů `x :: a`, `y :: b`. Nyní můžeme odvozovat typy výrazů na pravé straně definice: `(y, x) :: (b, a)` a `(x, y) :: (a, b)`.

Avšak návratový typ funkce musí být jednoznačný, a tedy oba typy si musí odpovídat: `(b, a)` ∼ `(a, b)`, z čehož vidíme, že oba prvky dvojice musí být stejného typu.

Celkový typ je tedy `mswap :: Bool -> (a, a) -> (a, a)`.

c) Funkci není možné otypovat, protože podle prvního vzoru je parametrem dvojice, podle druhého trojice a podle třetího čtveřice. Tyto tři typy však nejsou vzájemně unifikovatelné.

d) Funkce je syntakticky špatně zapsaná, protože jednotlivé definice mají různý počet argumentů. Nelze ji tedy otypovat.

Řešení 3.2.5

a) Číselný literál může být libovolného numerického typu, tedy `3 :: Num a => a`, `(+) :: Num b => b -> b -> b`. Dostáváme `Num a => a` ∼ `Num b => b`, z čehož dostáváme `a` ∼ `b` a typový kontext se nemusí rozšiřovat. Celkově tedy `(+ 3) :: Num a => a -> a`

b) Desetinný číselný literál je typu `3.0 :: Fractional a => a`, `(+) :: Num b => b -> b -> b`. Dostáváme tedy unifikaci `Num b => b` ∼ `Fractional a => a`, z čehož dostáváme `a` ∼ `b`, avšak zároveň nesmíme zapomenout na to, že obě proměnné mají nyní oba typové kontexty. Celkový typ je tedy `(+ 3.0) :: (Fractional a, Num a) => a -> a`.

Poznámka: Ve skutečnosti je ve standardní knihovně řečeno, že každý typ, který splňuje `Fractional`, nutně splňuje i `Num`, a tedy lze `Num` v tomto případě vynechat: `(+ 3.0) :: Fractional a => a -> a`. Jeho nevynechání však není chyba (nicméně interpretr jej automaticky vynechává).

c) Typy použitých podvýrazů jsou: `filter :: (a -> Bool) -> [a] -> [a]`, `(>=) :: Ord b => b -> b -> Bool`, `2 :: Num c => c`.

Nejprve určeme typ `(>= 2)`. Aplikací `2` jako druhého argumentu dostáváme unifikaci `Ord b => b` ∼ `Num c => c`. Pro typ `(>= 2)` tedy dostáváme dosazením do `Ord b => b -> b -> Bool` typ `(Ord b, Num b) => b -> Bool`.

Teď určíme typ celého výrazu. Aplikace funkce `filter` na `(>= 2)` nám vynucuje `a -> Bool` ∼ `(Ord b, Num b) => b -> Bool`. Tedy `a` ∼ `b` (plus typové kontexty). Hledaným typem je `[a] -> [a]`, což dosazením na základě výše určených informací dává výsledný typ `(Num a, Ord a) => [a] -> [a]`.

d) Typy použitých podvýrazů jsou: `2 :: Num a => a`, `(>) :: Ord b => b -> b -> Bool`, `div :: Integral c => c -> c -> c`, `3 :: Num d => d`, `(.) :: (f -> g) -> (e -> f) -> e -> g`.

Z aplikace `(> 2)` dostáváme unifikaci `Num a => a` ∼ `Ord b => b`, celkově je tedy výraz typu `(> 2) :: (Num a, Ord a) => a -> Bool`.

Z aplikace ``(`div` 3)`` dostáváme `Integral c => c` ∼ `Num d => d`, a tedy ``(`div` 3) :: (Integral c, Num c) => c -> c``.

Nyní, z použití ve skládání funkcí, dostáváme unifikaci `(Num a, Ord a) => a -> Bool` ∼ `(f -> g)`, a tedy `(Num a, Ord a) => a` ∼ `f`, `Bool` ∼ `g`. Dále potom `(Integral c, Num c) => c -> c` ∼ `e -> f`, a tedy `(Integral c, Num c) => c` ∼ `e` ∼ `f`. Nyní máme pro `f` dvě unifikace a musíme je dát dohromady: `(Integral c, Num c) => c` ∼ `(Num a, Ord a) => a` ∼ `f`.

Celkově dostaneme typ odpovídající `e -> g` (z definice `(.)`). Pro jednoduchost bereme lexikograficky první proměnnou, pokud může unifikace probíhat oběma směry:
`(> 2) . (`div` 3) :: (Num a, Integral a, Ord a) => a -> Bool`.

Poznámka: To lze dále zjednodušit (protože `Integral` vynucuje `Num` a `Ord`) na `(> 2) . (`div` 3) :: Integral a => a -> Bool`

Řešení 3.2.6

a) Výraz `id const` se vyhodnotí na výraz `const` a má tedy stejný typ.
`id const :: a -> b -> a`

b) Funkce `takeWhile` musí dostat 2 parametry -- funkci a seznam. Jelikož na prvky seznamu se bude aplikovat funkce `(even . fst)`, musí být tyto prvky uspořádané dvojice (aby bylo možno aplikovat na ně `fst`), jejichž první složka musí být celé číslo (přesněji být v typové třídě `Integral`, aby bylo možno na ni aplikovat `even`). Víme, že `takeWhile` vrací seznam stejného typu, jako seznam, který bere.
`takeWhile (even . fst) :: Integral a => [(a, b)] -> [(a, b)]`

c) Na vstupu musí být uspořádaná dvojice (abychom mohli aplikovat `snd`), druhou složkou které musí být opět uspořádaná dvojice (abychom pak mohli aplikovat `fst`).
`fst . snd :: (a, (b, c)) -> b`

d) Tenhle případ je analogický předešlému, jenom má více stupňů.
`fst . snd . fst . snd . fst . snd :: (a,((b,((c,(d,e)),f)),g)) -> d`

e) Na první argument budeme nejdříve aplikovat funkci `snd`, musí tedy jít o uspořádanou dvojici. Druhou složkou této dvojice musí být unární funkce, jelikož ta je použita jako první argument funkce `map`. Druhým argumentem je pak seznam, jehož prvky mají stejný typ, jaký vyžaduje zmíněná unární funkce. Výsledkem bude seznam prvků po aplikaci této unární funkce.
`map . snd :: (a, b -> c) -> [b] -> [c]`

f) Opět podobná úvaha: musí jít o uspořádanou dvojici, kde na druhou složku je možno aplikovat funkci `head` (je to tedy seznam). Na jeho první prvek je opět možné aplikovat `head`, prvky tohoto seznamu jsou tedy opět seznamy.
`head . head . snd :: (a, [[b]]) -> b`

g) Výraz vezme seznam a vrátí seznam, kde na každý prvek bude aplikována funkce `filter fst`. Tyto prvky musí být tedy opět seznamy (protože se na ně aplikuje `filter` podle predikátu `fst`). Prvky těchto vnitřních seznamů pak musí být uspořádané dvojice, jelikož na ně aplikujeme `fst`. A první složkou musí být `Bool`, protože ta je použita přímo jako predikát funkce `filter`.
`map (filter fst) :: [[(Bool, a)]] -> [[(Bool, a)]]`

h) Výraz vezme dva seznamy a vrátí třetí seznam (vychází z typu funkce `zipWith`). Prvek prvního a prvek druhého seznamu musí tvořit vhodné argumenty pro spojovací funkci `map`. První seznam tedy obsahuje nějaké unární funkce a druhý seznamy, kterých prvky je možno zpracovávat těmito unárními funkcemi. Výsledky aplikací zmíněných funkcí na seznamy v druhém seznamu jsou vráceny ve formě seznamu.
`zipWith map :: [a -> b] -> [[a]] -> [[b]]`

Řešení 3.2.7

a)
```
f1 :: a -> (a -> b) -> (a, b)
f1 x g = (x, g x)
```
b)
```
f2 :: [a] -> (a -> b) -> (a, b)
f2 s g = (h, g h) where h = head s
```
c)
```
f3 :: (a -> b) -> (a -> b) -> a -> b
f3 f g x -> head [f x, g x]
```
d)
```
f4 :: [a] -> [a -> b] -> [b]
f4 = zipWith (flip id)
```
e)
```
f5 :: ((a -> b) -> b) -> (a -> b) -> b
f5 g f  = head [g f, f undefined]
f5' g f = head [g f, f arg]
    where arg = arg
```
f)
```
f6 :: (a -> b) -> ((a -> b) -> a) -> b
f6 x y = x (y x)
```

Řešení 3.2.8 U prvního výrazu je každé `id` samostatnou instancí, tedy má svůj vlastní typ: první je typu `(a -> a) -> a -> a`, zatímco druhé je typu `b -> b`. Podobná situace je u druhého výrazu, jenom místo `id` používáme pro stejnou funkci pojmenování `f`.

Ve třetím výrazu je `id` argumentem s formálním jménem `x`, který však musí mít jeden konkrétní typ. Problém nastává v určování typu výrazu `f`: `f x = x x` -- uvažujme, že `x :: a`. Aplikace na pravé straně nás ale nutí specializovat, tedy `x :: a1 -> a2`. Jelikož je však `x` aplikováno samo na sebe, dostáváme typovou rovnost `a1 = a1 -> a2`, která vytváří nekonečný typ. Třetí výraz tedy není otypovatelný, a tudíž ani korektní.

Řešení 3.2.9 Nejdříve jsou uvedeny typy jednotlivých výrazů (případně výskytů požadované funkce), na posledním řádku za implikační šipkou se pak nachází výsledný typ, tedy typ průniku předchozích. Typové proměnné v jednotlivých řádcích spolu nejsou nijak provázané.

a) `[a] -> b`
`[a -> a] -> (b, c)`
⇒ `[a -> a] -> (b, c)`

b) `([[a]], b) -> c`
`(a, b) -> c`
⇒ `([[a]], b) -> c`

c) `(Num a) => a -> b`
`(Num c) => (a -> c, b)`
⇒ Typy jsou nekompatibilní, průnik je prázdný.

d) `(Num a) => [(a, b)]`
`[Char]`
⇒ Typy jsou nekompatibilní, průnik je prázdný.

e) `[a] -> b`
`(Num b) => [[a]] -> b`
⇒ `(Num b) => [[a]] -> b`

f) `(Num a) => a -> b`
`a -> (b -> b) -> c`
⇒ `(Num a) => a -> (b -> b) -> c`

Řešení 3.3.1 Lze nalézt v oficiální dokumentaci: `http://hackage.haskell.org/package/base/docs/Prelude.html#v:and`.

Řešení 3.3.2 `http://hackage.haskell.org/package/base/docs/Prelude.html#v:takeWhile`

Řešení 3.3.3

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 3.3.4 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 3.3.5

```
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 3.3.6

a) $\rightsquigarrow^*$ `[1,4,27,256,3125]`
b) ≡ `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 3.3.7 `zip = zipWith (,)`

Řešení 3.3.8

```
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 4.1.1

a) Korektní, není nutné použít všechny argumenty.
b) Korektní, argumentem může být i funkce, kterou budeme následně volat na nějakém argumentu.
c) Nekorektní, platnost λ-abstrakce končí v místě čárky ukončující první člen uspořádané dvojice. Ve druhé složce uspořádané dvojice již `s` není definované (jestli není definované v nadřazeném kontextu).
d) Korektní, λ-abstrakce je možné zanořovat i s argumenty. V tomhle případě je implicitní závorkování následovné:

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

λ-abstrakce totiž končí nejdále jak je to syntakticky možné.
e) Korektní, argumentem λ-abstrakce nemusí být jenom jednoduchá proměnná: `[(x, y)]` představuje vzor pro jednoprvkový seznam obsahující uspořádanou dvojici. Aplikování λ-abstrakce na argumenty jiného tvaru nebo typu selže.
f) Korektní, ekvivalentní s `_ [_] -> ()`. Ignoruje obsah prvních dvou argumentů, avšak vynucuje, že druhým argumentem je jednoprvkový seznam.
g) Nekorektní, argumenty musí být unikátní, tedy není možné použít jeden formální argument před `->` vícekrát.
h) Korektní, výraz je ekvivalentní výrazu `($) ($) ($)`.
i) Korektní, nedochází k nevhodnému aplikování výrazů a použité proměnné jsou vždy definovány v některé λ-abstrakci.
j) Korektní, funkce bere jako argument seznam (který však musí být prázdný) a vrací `()`.

Řešení 4.1.2

a) Ne, `x` je v dané operátorové sekci argumentem. Funkce `flip` musí mít jako první argument vždy funkci, tedy `(<)`. `(<x)` je jenom zkrácený zápis pro `\y -> y < x`. Správně by tedy bylo `\x -> flip (<) x`.
b) Ne, správné implicitní závorkování je `((.) f) (g x)`. Správnou úpravou by tedy bylo `\x -> ((.) f . g) x` nebo `\x -> f . g x`.
c) Ne, u operátorových sekcí jako je `(.g)` není možné udělat opak η-redukce `(.g)` $\rightsquigarrow$ `\x -> (.g x)`. Správná úprava by byla na `\x -> f ((.g) x)` $\rightsquigarrow$ `\x -> f (x . g)`.
d) Ano, implicitní závorkování částečné aplikace je `(((const (+)) x) y) z`. Argument `x` již na pravé straně sice nepoužíváme, nemůžeme jej však odstranit z formálních parametrů, neboť celkový typ výrazu by se změnil.
e) Ano, přepis je přesně podle definice operátorové sekce. Poznamenejme ještě, že přepis na `_ -> 3 + 2` by nebyl správný (nikdo nezaručuje, že operátor `(+)` je skutečně komutativní -- můžeme si jej třeba předefinovat).

Řešení 4.1.3 Žádný, přesouvání argumentů do vnitřních λ-abstrakcí je ekvivalentní pohledu na funkci skrz částečnou aplikaci.

Řešení 4.1.4

a) `map (+) [1, 2, 3]`
b) `\t u -> t * u 2 3`
V tomto případě není co aplikovat, protože celý zbytek výrazu za šipkou je součástí těla λ-abstrakce.
c) Výraz není korektní. Podvýraz `(t * u 2)` se vyhodnotí na číslo a ten bychom následně aplikovali na číslo `3`, což samozřejmě nelze.
d) `4 + 0.3`
e) `(3 + 1) * 10`
Myšlenkou je, že netřeba zapomenout na zachování závorek po dosazení.
f) `const map (head . head) filter`
Opět, nezapomínat na závorky.
g) `zipWith (\x y -> x * 10 + y) [1..10] (map (^2) [1..5])`
h) Vyhodnocování rozepíšeme podrobněji:

```
(\x y -> x (map y)) (\s (a, b) -> s [a..b]) (\f -> f - 1)
(\y -> (\s (a, b) -> s [a..b]) (map y)) (\f -> f - 1)
(\s (a, b) -> s [a..b]) (map (\f -> f - 1))
\(a, b) -> map (\f -> f - 1) [a..b]
```

Řešení 4.2.1

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 -> 3 + 60 `div` x ^ 2 > 0) 7`

```
\x -> 3 + 60 `div` x ^ 2 > 0
\x -> (3 + (60 `div` (x ^ 2))) > 0
\x -> (>0) ((3+) (div 60 ((^2) x)))
\x -> ((>0) . (3+) . div 60 . (^2)) x
(>0) . (3+) . div 60 . (^2)
```

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

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

nebo

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

e) `(\s -> "<" ++ s ++ ">") "boxxy"`

```
\s -> "<" ++ s ++ ">"
\s -> "<" ++ (s ++ ">")
\s -> ("<"++) ((++">") s)
\s -> (("<"++) . (++">")) s
("<"++) . (++">")
```

Poznamenejme, že funkce `(++">") . ("<"++)` sice představuje ve výsledku stejnou úpravu, avšak není správním výsledkem, protože by vznikla uzávorkováním `("<" ++ s) ++ ">"`, které ale není korektní z hlediska asociativity `(++)`. Dalším důvodem je, že vyhodnocení by zabralo jiný počet (více) kroků.

f) `(\x -> 0 < 35 - 3 * 2 ^ x) 2.5`

```
\x -> 0 < 35 - 3 * 2 ^ x
\x -> 0 < (35 - (3 * (2 ^ x)))
\x -> (0<) ((35-) ((3*) ((2^) x)))
\x -> ((0<) . (35-) . (3*) . (2^)) x
(0<) . (35-) . (3*) . (2^)
```

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

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

h) `(\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.

i) `(\x y -> 2 * x + y) (sin 1000) (sum [1..1000])`

```
\x y -> 2 * x + y
\x y -> (+) (2 * x) y
\x -> (+) (2 * x)
\x -> (+) ((2*) x)
\x -> ((+) . (2*)) x
(+) . (2*)
```

Ačkoliv poslední výraz vypadá možná nelogicky, vzpomeňte si na to, že při skládání funkcí považujeme oba argumenty za unární funkce, a tedy `(+)` bereme jako funkci, která požaduje jeden číselný argument a vrátí „přičitovací“ funkci.

Řešení 4.2.2

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 4.2.3

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)
```

e) `flip flip 0`
```
\f -> flip flip 0 f
\f -> flip f 0
\f x -> flip f 0 x
\f x -> f x 0
```

f) `(.) (+) . (+)`
```
\a -> ((.) (+) . (+)) a
\a -> (.) (+) ((+) a)
\a -> (+) . ((+) a)
\a -> (+) . (+) a
\a b -> ((+) . (+) a) b
\a b -> (+) ((+) a b)
\a b c -> (+) ((+) a b) c
\a b c -> a + b + c
```
(Závorky v posledním kroku je možno odstranit, protože `(+)` je asociativní zleva.)

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

Řešení 4.2.4

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 4.2.5

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 4.2.6 Několikrát po sobě použijeme funkci `flip`.

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

Řešení 4.2.7 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 4.2.8

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 4.2.9

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

```
nebo :: (Bool, Bool) -> Bool
nebo (x, y) = x || y
```

`curry nebo` ≡ `(||)`
`uncurry (||)` ≡ `nebo`

Řešení 4.3.2

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x, y, z)

uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
uncurry3 f (x, y, z) = f x y z
```

Řešení 4.3.3 Ne. Dvouargumentové funkce pracují s uspořádanými dvojicemi a tříargumentové funkce s uspořádanými trojicemi. Uspořádané dvojice a trojice však mezi sebou nemají žádný speciální vztah.

Řešení 4.3.4

a)
```
\(x, y) -> x + y
\(x, y) -> (+) x y
\(x, y) -> uncurry (+) (x, y)
uncurry (+)
```

b)
```
\x y -> nebo (x, y)
\x y -> curry nebo x y
curry nebo
```

c)
```
\((x, y), z) -> x + y + z
\((x, y), z) -> (x + y) + z
\((x, y), z) -> ((+) x y) + z
\((x, y), z) -> (uncurry (+) (x, y)) + z
\((x, y), z) -> (+) (uncurry (+) (x, y)) z
\((x, y), z) -> ((+) . uncurry (+)) (x, y) z
\((x, y), z) -> uncurry ((+) . uncurry (+)) ((x, y), z)
uncurry ((+) . uncurry (+))
```

Řešení 4.3.5

a)
```
dist (curry id) id
\x -> dist (curry id) id x
\x -> (curry id) x (id x)
\x -> ((\f x y -> f (x, y)) id) x x
\x -> (\f x y -> f (x, y)) id x x
\x -> id (x, x)
\x -> (x, x)
```

b)
```
uncurry (dist . ((.) (curry id)))
(\f (x, y) -> f x y) (dist . ((.) (curry id)))
\(u, v) -> (\f (x, y) -> f x y) (dist . ((.) (curry id))) (u, v)
\(u, v) -> (dist . ((.) (curry id))) u v
\(u, v) -> ((dist . ((.) (curry id))) u) v
\(u, v) -> (dist (((.) (curry id)) u)) v
\(u, v) -> dist (((.) (curry id)) u) v
\(u, v) w -> dist (((.) (curry id)) u) v w
```

```
\(u, v) w -> (\f g x -> f x (g x)) (((.) (curry id)) u) v w
\(u, v) w -> (((.) (curry id)) u) w (v w)
\(u, v) w -> (.) (curry id) u w (v w)
\(u, v) w -> ((.) (curry id) u w) (v w)
\(u, v) w -> ((curry id . u) w) (v w)
\(u, v) w -> (curry id . u) w (v w)
\(u, v) w -> ((\f x y -> f (x, y)) id . u) w (v w)
\(u, v) w -> ((\x y -> (x, y)) . u) w (v w)
\(u, v) w -> (((\x y -> (x, y)) . u) w) (v w)
\(u, v) w -> ((\x y -> (x, y)) (u w)) (v w)
\(u, v) w -> (\x y -> (x, y)) (u w) (v w)
\(u, v) w -> (u w, v w)
```

Řešení 5.1.1

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 je zas vrácen výsledek nějaké funkce, která bere jako argument první prvek a rekurzivní volání stejné funkce (i s případnými argumenty jako u `map'`) na zbytku seznamu.

Jejich funkcionalitu 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)
```

Řešení 5.1.2

a) Funkce vloží mezi každý člen seznamu operátor `+` a na konec seznamu zařadí `0`. Tedy lze ji aplikovat na číselný seznam a výsledkem je součet hodnot jeho prvků, tj. odpovídá funkci `sum`.

b) 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]` $\leadsto^*$ `1 + 10 * (2 + 10 * 3)`
   $\leadsto^*$ `321`

c) 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.

Řešení 5.1.3 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 (-)
```

Řešení 5.1.4

a) `compose = foldr (.) id`
   V tomto případě by fungovalo i `compose = foldl (.) id`, protože operace skládání je asociativní. Ale přirozenou asociativitou pro ni je asociativita zprava, proto jsme použili `foldr`.
b) Viz příklad 5.1.5.

Řešení 5.1.5

a) Funkce `foldr`, jak je nám známo, 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 typ 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`

Řešení 5.1.6 Nejprve si výraz upravíme na `\z s -> foldr (:) z s`. Vzpomeňme si, že `foldr` nahrazuje výskyty `(:)` a `[]`. V tomto případě výskyty `(:)` neměníme. Nahradíme jenom výskyt prázdného seznamu na konci. Intuitivně, prázdný seznam můžeme nahradit libovolným seznamem s prvky stejného typu, jako má vstupní seznam.

Když si pak představíme, co bude takováto funkce vracet, bude to seznam obsahující nejprve prvky seznamového argumentu funkce `foldr` a pak na místo `[]` dosadíme seznam `z`. Tedy výraz `foldr (:) z s` vrací stejný výsledek jako výraz `s ++ z`. Je tedy ekvivalentní výrazu `flip (++)`.

Po odvození typu výrazu dostaneme `[a] -> [a] -> [a]`.

Řešení 5.1.7 Víme, že funkce `foldl` „skládá“ seznam zleva. Podívejme se, jak dochází k vyhodnocování této funkce na krátkém seznamu:

```
   foldl (flip (:)) [] [1,2,3]
⇝ flip (:) (flip (:) (flip (:) [] 1) 2) 3
⇝ 3 : (flip (:) (flip (:) [] 1) 2)
⇝ 3 : (2 : (flip (:) [] 1))
⇝ 3 : (2 : (1 : []))
≡  [3,2,1]
```

Vidíme tedy, že tato funkce funguje jako funkce `reverse`.

Typ funkce je `[a] -> [a]`.

Řešení 5.1.8 Pro některé podúlohy je uvedeno více ekvivalentních řešení -- tato jsou uváděna pod sebou a jsou odlišena apostrofem v názvu funkce.

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

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

c)
```
productFold :: Num a => [a] -> a
productFold = foldr (*) 1
productFold' = foldr (\e t -> e * t) 1
```

d)
```
orFold :: [Bool] -> Bool
orFold = foldr (||) False
orFold' = foldr (\e t -> e || t) False
```

e)
```
andFold :: [Bool] -> Bool
andFold = foldr (&&) True
andFold' = foldr (\e t -> e && t) True
```

f)
```
minimumFold :: Ord a => [a] -> a
minimumFold = foldr1 min
minimumFold' = foldr1 (\e t -> if e < t then e else t)
minimumFold'' list = foldr min (head list) list
```

g)
```
maximumFold :: Ord a => [a] -> a
maximumFold = foldr1 max
maximumFold' = foldr1 (\e t -> if e < t then t else e)
maximumFold'' list = foldr max (head list) list
```

h)
```
maxminFold :: Ord a => [a] -> (a,a)
maxminFold list = foldr (\e (tMin, tMax) -> (min e tMin, max e tMax) )
   (head list, head list) list
```

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

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

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

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

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

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

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

p)
```
reverseFold :: [a] ->  [a]
reverseFold = foldl (flip (:)) []
reverseFold' = foldl (\t e -> e : t) []
reverseFold'' = foldr (\e t -> t ++ [e]) []
```

Poslední řešení má výrazně vyšší složitost!

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

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

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

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

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

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

Druhé uvedené řešení má lepší složitost.

Řešení 5.1.9

```
foldl f z s = foldr (flip f) z (reverse s)
```

Řešení 5.1.10 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)
```

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

Řešení 5.1.11 Funkce `foldr` přestane vyhodnocovat výraz po prvním nalezeném `True` (díky vlastnosti funkce `(||)`, která se nazývá „short-circuiting“), avšak `foldl` ho vždy projde celý. Tedy v případě nekonečného seznamu `foldr` skončí po prvním nalezeném `True`, ale `foldl` neskončí nikdy.

Řešení 5.1.12

```
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 5.1.10).

Řešení 5.2.1

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 5.2.2

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

Řešení 5.2.3

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 5.2.4 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 5.2.5

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

Řešení 5.2.6 `p = sum (zipWith (/) (iterate negate 4) [1,3..])`

Řešení 6.1.1 V prvním případě dojde jenom k vypsání hodnoty výrazu `"**\n**\n"` jako řetězce. Jelikož jde o výraz vytvořený jenom z datových konstruktorů (seznam znaků), nelze jej dále vyhodnotit.

Na druhou stranu, u výrazu `putStr "**\n**\n"` jde o výraz typu `IO ()`. To znamená, že výsledkem, na který se výraz vyhodnotí, bude nultice obalená monádou. Avšak, jako vedlejší efekt vyhodnocení akce `putStr` dojde k vypsání její argumentu na standardní výstup.

Poznámka: Srovnejte výsledek vyhodnocení následovných akcí v interpretu ghci:

```
putStrLn "test"
putStr ""
return ()
return True
return 100
return [1,2,3]
return (return 1 :: IO Int)
return id
return (id, not)
```

Na základě výsledku vyhodnocení těchto výrazů můžeme vidět, že interpret nevypíše hodnotu, kterou vracíme v monádě v případě, že jde o hodnotu typu, který není instancí třídy `Show`, protože ji vypsat nelze a také pokud jde o hodnotu `()`, protože ta reprezentuje v Haskellu hodnotu typu známého z jazyka C jako `void` a tudíž nenese žádnou užitečnou informaci. Navíc u hodnoty typu `IO ()` není výsledek vypisován patrně i proto, že by byl rušivý při výpisu hodnot pomocí funkcí `putStr`, `putStrLn`, `print`, `...`

Řešení 6.2.1

```
getInt :: IO Int
getInt = getLine >>= \num -> return (read num :: Int)
```

```
getIntDo :: IO Int
getIntDo = do
    line <- getLine
    let num = read line :: Int
    return num
```

Řešení 6.2.2

```
trg :: IO ()
trg = do
    putStrLn "Enter three numbers separated by commas: "
    nums <- getLine
    let (a, b, c) = read ("(" ++ nums ++ ")") :: (Int, Int, Int)
    print (a + b > c && b + c > a && c + a > b)
```

Řešení 6.2.3

```
import System.Directory

cat :: IO ()
cat = do    putStr "Enter filename: "
            fileName <- getLine
            fileExists <- doesFileExist fileName
            if fileExists then do
                fileContents <- readFile fileName
                putStr fileContents
            else do
                putStrLn "Requested file was not found."
                return ()
```

Řešení 6.2.4 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. Tedy rekurze umožňuje kontrolovaně (zadáme-li prázdný řetězec -- dojde k ukončení rekurze) opakovat akci. Pokud však neuhlídáme ukončení rekurze, může dojít k zacyklení vyhodnocování akcí.

Řešení 6.3.1

a) Program nejprve načte pomocí `getLine` od uživatele řetězec. Pak jej pomocí operátoru `(>>=)`, nazývaného také bind, předáme jako argument funkci `putStr . filter isAlpha`, která z řetězce zachová jenom písmenné znaky a následně výsledek vypíše na výstup.

b)
```
import Data.Char
main' :: IO ()
main' = do  x <- getLine
            putStr (filter isAlpha x)
```

Řešení 6.3.2

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

Řešení 6.3.3

a) Nekorektní, operátor je zapsaný v prefixovém tvaru, ale použitý infixově.
b) Korektní.
c) Nekorektní, funkci `return` chybí argument.
d) Nekorektní, není možné „sesbírat“ hodnoty `getLine` takovým způsobem (použití `>>` za prvním `getLine` způsobí „zapomenutí“ jeho hodnoty).
e) Nekorektní, platnost `s` končí závorkou, tedy při použití na konci řádku již není definována.
f) Korektní, proměnnou `f` není nutné použít.
g) Nekorektní, za operátorem `>>` musí následovat výraz typu `IO a`, ne funkce typu `a -> IO b`.
h) Nekorektní, zápis `<-` je možno použít jenom v `do`-notaci (a intensionálních seznamech).

Řešení 6.3.4

```
query' :: String -> IO Bool
query' question =
    putStrLn question >> getLine >>= \answer -> return (answer == "ano")
```

Nebo lze upravit vzniklou λ-abstrakci na pointfree tvar:

```
query'' :: String -> IO Bool
query'' question =
    putStrLn question >> getLine >>= return . (=="ano")
```

Řešení 6.3.5

a)
```
query2 :: String -> IO Bool
query2 question = do
    putStrLn question
    answer <- getLine
    if answer == "ano"
        then return True
        else if answer == "ne"
            return 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
        return True
```

```
        else
            if lcAnswer `elem` ["ne", "nie", "no"] then
                return False
            else
                query3 question
```

Řešení 6.3.8

```
x >> f = x >>= \_ -> f
```

Řešení 6.4.1

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

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

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

Řešení 6.4.2

```
data Jar = EmptyJar
   | Cucumbers
   | Jam String
   | Compote Int
   deriving (Show, Eq)
today :: Int
today = 2014
stale :: Jar -> Bool
stale EmptyJar = False
stale Cucumbers = False
stale (Jam _) = False
stale (Compote x) = today - x  >= 10
```

Řešení 6.4.3

a) Nulární typový konstruktor `X`, nulární datový konstruktor `X`.
b) Nulární typový konstruktor `A`, nulární datový konstruktor `X`, unární datový konstruktor `Y`, binární datový konstruktor `Z`.
c) Unární typový konstruktor `B` (konkrétní typ pak může být například `B Int` nebo `B [String]`). Nulární datový konstruktor `A`, unární datové konstruktory `B` a `C`.
d) Nulární typový konstruktor `C`, unární datový konstruktor `D`.
e) Nulární typový konstruktor `E`, unární datový konstruktor `E`.
f) Definice (nulárního) typového aliasu `String`.

Řešení 6.4.4

a) Nulární typový konstruktor `X`, unární datový konstruktor `Value`.
b) Unární typový konstruktor `X`, unární datový konstruktor `V`.
c) Nulární typový konstruktor `X`, ternární datový konstruktor `Test`.
d) Nulární typový konstruktor `X`, nulární datový konstruktor `X`.
e) Nulární typové konstruktory `M` a `N`, nulární datové konstruktory `A`, `B`, `C`, `D`, unární datové konstruktory `N` a `M`.
f) Unární typový konstruktor `Test`, binární datové konstruktory `F` a `M`.
g) Chybná deklarace: `Hah` je v seznamu použito jako typový konstruktor, jedná se však o datový konstruktor.
h) Nulární typový konstruktor `FNM`, ternární datový konstruktor `T`.
i) Chybná deklarace: chybí datový konstruktor.
j) Vytváří se pouze typové synonymum (nulární).

Řešení 6.4.5

a) Příklady hodnot jsou:

```
Kvadr 1 2 3
Kvadr (-4) 3 4.3
Valec 3 (1/2)
Koule (sin 2)
```

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

b) Datové konstruktory jsou umístěny jako první identifikátor ve výrazech oddělených svislítky. Tedy v tomto případě to jsou `Kvadr, Valec, Kuzel, Koule`. Také datový 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 `Teleso`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Float`.

d) Funkci budeme definovat po částech. Pro každý možný tvar hodnoty typu `Teleso`, 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.

```
objem :: Teleso -> Float
objem (Kvadr x y z) = x * y * z
objem (Valec r v)   = pi * r * r * v
...

povrch :: Teleso -> Float
povrch (Kvadr x y z) = 2 * (x * y + x * z + y * z)
povrch (Valec r v)   = 2 * pi * r * (v + r)
...
```

e)
```
import Thinking.Fantasy

-- TODO
```

Řešení 6.4.6

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
```

Řešení 6.4.8

a) 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
```

b) 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 téhle rovnosti.

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

c) Zde opět efektivně použijeme vestavěnou funkci `signum`.

```
nonneg :: Frac -> Bool
nonneg (a,b) = signum (a*b) >= 0
```

d) K řešení nám opět pomůže nejdříve si 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)
```

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

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

g)
```
fracdiv :: Frac -> Frac -> Frac
fracdiv (_,_) (0,_) = error "division by zero"
fracdiv (a,b) (c,d) = fractimes (a,b) (d,c)
```

h) Pro výpočet průměru musíme nejdříve určit součet všech zlomků v seznamu. To zjistíme kombinací akumulační funkce `foldr1` a součtu zlomků. Následně součet vydělíme délkou seznamu -- jestliže však chceme použít funkci na dělení zlomků, kterou jsme definovali dřív, musíme délku převést na zlomek.

```
fracmean :: [Frac] -> Frac
fracmean s = fracdiv (foldr1 fracplus s) (length s,1)
```

Poznámka: Haskell má vestavěný typ pro zlomky, `Rational`, ten je reprezentován v podstatě stejným způsobem, jako dvě hodnoty typu `Integer`. Nicméně nejedná se o dvojice, typ `Rational` definuje datový 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 `(+), (-), (*), (/)`.

Řešení 6.4.9

a) Ok, datové a typové konstruktory můžou mít stejné názvy.
b) Nok, chybí datový konstruktor.
c) Ok, jednoduché typové synonymum.
d) Ok.
e) Ok.
f) Nok, typová proměnná `a` musí být argumentem konstruktoru `Makro`.
g) Nok, datový konstruktor není možné použít vícekrát.
h) Nok, typová proměnná `c` musí být argumentem konstruktoru `Fun`.
i) Nok, argumenty konstruktoru `Fun` mohou být pouze typové proměnné, ne složitější typové výrazy (tj. není možné použít definici podle vzoru).
j) Ok, typové synonymum nemusí být lineární ve svých argumentech (nemusí být každý použit právě jednou).
k) Nok, `Z X` není korektní výraz, protože `X` je datový konstruktor.
l) Nok, v argumentech datových konstruktorů se při deklaraci mohou vyskytovat pouze typy, ne datové konstruktory.
m) Nok, každý datový konstruktor musí začínat velkým písmenem.
n) Nok, výraz je interpretován jako datový konstruktor `Makro` se třemi argumenty: `Int`, `->` a `Int` -- je nutné přidat závorky kolem `(Int -> Int)`.
o) Nok, syntax výrazu je chybná: `type` musí mít na pravé straně pouze jednu možnost (jedná se o typové synonymum, ne o nový datový typ).
p) Ok, i když neexistuje žádná konečná úplně definovaná hodnota. Hodnotou tohoto typu je například `x = Value x`.
q) Nok, není možné použít stejný datový konstruktor ve více typech.
r) Nok, chyba je v definici `type`: `GoodChoice` je datový konstruktor, tj. není možné, aby výsledkem byla jenom část typu (hodnoty typu `Choice t1 t2` vytvořené pomocí datového konstruktoru `GoodChoice`).
s) Ok, nepřímo rekurzivní datový typ je v pořádku.
t) Ok, typový i datové konstruktory mají stejný název. Na pravé straně definice je `X` nejdřív binárním datovým a pak dvakrát nulárním typovým konstruktorem. Jediná plně definovaná hodnota tohoto typu je `x = X x x`.

Řešení 6.4.10

a) Chybná hodnota, `String` je typ a není možné na něj aplikovat datový konstruktor.
b) `Num a => T a`
c) `T String`
d) Chybná hodnota, `T` není datový konstruktor.
e) `A Char`

f) `A a`
g) `Num a => A a`
h) `x :: A a`
i) Chybná hodnota, `N A :: M`, `M :: N -> N`.
j) `N`
k) `X Bool a`
l) `X a b -> X a b`
m) `(Num a, Num b) => [X [b] (Maybe a)]`
n) `a -> X b a`
o) `X (a -> X b a) c`

Řešení 7.1.1 Typová třída umožňuje deklarovat určitou vlastnost datového typu na základě definice několika funkcí. Například to, že je na datovém typu definována rovnost, uspořádání a nebo že jeho hodnoty je možné převést do řetězcové reprezentace. Výhodou používání typových tříd je možnost používat stejné intuitivní operátory nebo funkce jako pro jiné typy a taky možnost získat na základě definovaných základních funkcí i další funkce, které lze vyjádřit pomocí základních. U některých jednoduchých případů datových typů a jednodušších typových tříd je možné i automaticky odvodit definice základních funkcí (!), k čemuž se používá klauzule `deriving` následována n-ticí typových tříd, pro které mají být příslušné funkce automaticky odvozeny.

Řešení 7.1.2

```
instance Show TrafficLight where
  show Red       = "Red light!"
  show Orange    = "Orange light!"
  show Green     = "Green light!"
instance Eq TrafficLight where
  Red        == Red       = True
  Orange     == Orange    = True
  Green      == Green     = True
  _          == _         = False
instance Ord TrafficLight where
  _          <= Red       = True
  Orange     <= Orange    = True
  Green      <= _         = True
  _          <= _         = False
```

Řešení 7.1.3

```
instance (Eq a, Eq b) => Eq (PairT a b) where
  (PairD x y) == (PairD v w) = (x == v) && (y == w)
instance (Show a, Show b) => Show (PairT a b) where
  show (PairD x y) = "pair of " ++ show x ++ " and " ++ show y
instance (Ord a, Ord b) => Ord (PairT a b) where
  (PairD x y) <= (PairD v w) = x < v || (x == v && y <= w)
```

Řešení 7.1.4

```
iff :: Boolable b => b -> a -> a -> a
iff b t e = if getBool b then t else e

class Boolable a where
  getBool :: a -> Bool

instance Boolable Bool where
  getBool x = x

instance Boolable Int where
  getBool 0 = False
  getBool _ = True

instance Boolable [a] where
  getBool [] = False
  getBool _ = True
```

Řešení 7.1.5 V prvním případu jde o zavedení operátoru `(<=)` na datovém typu `Nat`. Tento operátor bude tedy možné používat na typech jako doposud plus na typu `Nat`.

Ve druhém případě jde o redefinici operátoru `(<=)`. Tento operátor bude tedy možné používat jenom na typu `Nat`. Původní definice bude překryta touto a nebude dostupná přímo. (Ale bude dostupná při uvedení kvalifikovaného názvu operátoru: `(Prelude.<=) 3 4` nebo `3 Prelude.<= 4`)

Řešení 7.2.1

a) Nekorektní výraz -- typový konstruktor `Maybe` aplikovaný na hodnotu.
b) Korektní typ s hodnotou například `Just 0`.
c) Korektní hodnota typu `Maybe t` v případě, že `a` je korektní hodnota (definovaná externě).
d) Nekorektní výraz -- datový 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 datový 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, typ je uvedený v zadání.
k) Nekorektní výraz -- `Just Nothing` je typu `Maybe (Maybe a)`. Typ uvedený v zadání je všeobecnější než skutečný nejvšeobecnější typ výrazu.
l) Nekorektní výraz -- typový konstruktor `Maybe` obsahující uvnitř datový konstruktor `Just`. Dodejme, že `Maybe [a -> Maybe Char]` by byl korektní typ.
m) Korektní hodnota typu `(Num a) => Maybe (a -> a)`.
n) Nekorektní výraz -- hodnota typu `(Num a) => Maybe (a -> a)` (která není funkcí) je aplikována na hodnotu typu `(Num b) => Maybe b`.
o) Korektní hodnota typu `Bool -> Maybe a -> Maybe a`.

p) Korektní hodnota (funkce) typu `a -> Maybe a`.
q) Korektní hodnota (ne funkce) typu `Maybe (a -> Maybe a)`.
r) 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.
s) Nekorektní výraz -- důvody jsou poněkud složitější. Typový konstruktor `Maybe` je druhu `* -> *`, tedy akceptuje typy druhu `*` a vrací typ s druhem `*`. Argument však nemá správný druh (substituci jako u typů není možné použít, druhy nejsou polymorfní).
t) Nekorektní výraz -- typový kontext se udává pro celý typ, nikdy nesmí být zanořený uvnitř typu.

Řešení 7.2.2

```
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
```

Řešení 7.2.3 Nejprve vyřešme podmínku, kdy může taková instance existovat. Jinak řečeno, jaké podmínky jsou kladeny na datový typ `a`. Když budeme chtít porovnávat hodnoty typu `MyMaybe a`, budeme potřebovat porovnávat i hodnoty typu `a`. To tedy znamená, že `a` musí být instancí třídy `Ord`.

Ještě musíme dodefinovat nové uspořádání. Snad nejpřirozenějším (ale ne jediným možným!) uspořádáním je takové, které považuje `MyNothing` za nejmenší prvek a všechny hodnoty tvaru `MyJust x` za větší než `MyNothing`, přičemž uspořádání na hodnotách tvaru `MyJust x` je převzato z typu `a`.

Abychom mohli definovat typ jako instanci typové třídy, musíme k tomu definovat minimální množinu funkcí, která je pro danou třídu potřebná. Pro třídu `Ord` je to například množina {`(<=)`}. Ve výsledku tedy instanciace může vypadat následovně:

```
instance Ord a => Ord (MyMaybe a) where
    MyNothing <= _        = True
    MyJust x  <= MyJust y = x <= y
    _         <= _        = False
```

Řešení 7.3.1

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

```
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.

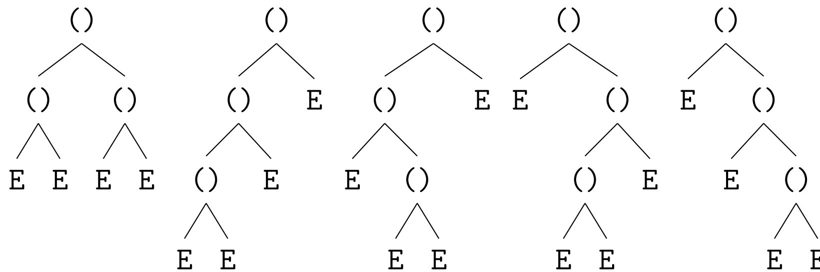
d)
```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Succ x) = 1 + natToInt x
```

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

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

Řešení 7.3.2

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

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

c)

$$\#_{\texttt{Bool}}(n) = 2^n \#_{()}(n)$$

Obecně pro `BinTree` t máme:

$$\#_t(n) = |t|^n \#_{()}(n),$$

kde $|t|$ je počet různých hodnot typu t.

d) Budeme postupovat rekurzivně vůči struktuře stromu. Strom může být buď tvaru `Empty` nebo tvaru `Node x l r`. V prvním případě je počet uzlů 0. V druhém případě je počet uzlů součtem počtu uzlů podstromů `l` a `r` plus jedna (za uzel samotný). Zapsáno vypadá definice následovně:

```
size :: BinTree a -> Int
size Empty        = 0
size (Node x l r) = 1 + size l + size r
```

Řešení 7.3.3

```
treeSize :: BinTree a -> Integer
treeSize Empty          = 0
treeSize (Node _ t1 t2) = 1 + treeSize t1 + treeSize t2

treeMax :: Ord a => BinTree a -> a
treeMax (Node v Empty Empty) = v
treeMax (Node v t     Empty) = max v (treeMax t)
treeMax (Node v Empty t    ) = max v (treeMax t)
treeMax (Node v t1    t2   ) = maximum [v, treeMax t1, treeMax t2]
treeMax Empty                = error "treeMax: Empty tree"

listTree :: BinTree a -> [a]
listTree Empty          = []
listTree (Node v t1 t2) = listTree t1 ++ [v] ++ listTree t2

treeMax' :: Ord a => BinTree a -> a
treeMax' = maximum . listTree

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
```

Řešení 7.3.4

a)
```
fullTree :: Int -> a -> BinTree a
fullTree 0 _ = Empty
fullTree n v = Node v (fullTree (n-1) v) (fullTree (n-1) v)
```

b)
```
height :: BinTree a -> Int
height Empty        = 0
height (Node x l r) = 1 + max (height l) (height r)
```

c)
```
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
```

Řešení 7.3.5

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

b)
```
nilTree :: BinTree [a]
nilTree = treeRepeat []
```

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

Řešení 7.3.6

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

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

Řešení 7.3.7

```
bt1 = Node 0 bt1 bt1
bt2 n = Node n sub sub where sub = bt2 (n + 1)
bt3 = Node "strom" bt3 Empty
bt4 = iterate (Node 0 Empty) Empty
bt5 = until (const False) (Node 0 Empty) Empty
```

Poznamenejme, že `bt4` není úplně přesné řešení, protože je to seznam konečných binárních stromů, kterého teoretickým posledním prvkem je požadovaný strom. Funkce `until` je standardně definována v Prelude následovně:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x
    | p x       = x
    | otherwise = until p f (f x)
```

Řešení 7.3.8

```
ntreeSize :: NTree a -> Integer
ntreeSize (NTree _ subtrees) = 1 + sum (map ntreeSize subtrees)

ntreeSum :: NTree a -> Integer
ntreeSum (NTree v subtrees) = v + sum (map ntreeSum subtrees)

instance Eq a => Eq (NTree a) where
    NTree x sub1 == NTree y sub2 = x == y && sub1 == sub2

-- poznamka: vyuzivame toho, ze je-li a instanci Eq/Ord pak i [a] je
   instanci prislusne tridy

instance Ord a => Ord (NTree a) where
    NTree x sub1 <= NTree y sub2 = x < y || (x == y && sub1 <= sub2)

ntreeMap :: (a -> b) -> NTree a -> NTree b
ntreeMap f (NTree v subtrees) = NTree (f v) (map (ntreeMap f) subtrees)
```

Řešení 8.1.1

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

Řešení 8.1.2

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 8.1.3 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 8.1.4

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..] ]`
f) `[ [n | _ <- [1..(2*n-1)]] | n <- [1..] ]`
g) `[ [c | _ <- [1..k] ] | (k, c) <- zip [1..26] ['z','y'..'a'] ]`
h) `[ [ [1|_<-[1..n]] | _ <- [1..n]] | n <- [1..] ]`

Řešení 8.1.5

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 8.1.6 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 8.1.7

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 8.1.8 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 8.1.9

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]
```

Řešení 8.2.1 Nejprve je třeba přemyslet si, jak by taková funkce intuitivně měla fungovat. Katamorfismus je obecně funkce na struktuře, která nahrazuje konstruktory této struktury funkcemi, a ve výsledku umožní vyhodnocení nebo její „kolaps“ na jedinou hodnotu. 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 datové 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`. `Zero` nahradíme nulární funkcí, tedy hodnotou typu `a`. `Succ` zas nahradíme unární funkcí s typem `a -> a`. Když definujeme tuto transformaci jako funkci, musíme ji definovat po částech pro jednotlivé datové 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 (nfold 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 (nfoldsz x)
```

Řešení 8.2.2

a)
```
treeSum :: Num a => BinTree a -> a
treeSum = treeFold (\v l r -> v + l + r) 0
```

b)
```
treeProduct :: Num a => BinTree a -> a
treeProduct = treeFold (\v l r -> v * l * r) 1
```

c)
```
treeOr :: BinTree Bool -> Bool
treeOr = treeFold (\v l r -> v || l || r) False
```

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

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

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

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

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

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

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

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

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

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

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

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

p)
```
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
```

q)
```
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
```

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

s)
```
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
```

t)
```
findPredicates :: a -> BinTree (Int, a -> Bool) -> [Int]
findPredicates x = treeFold (\(n,v) l r -> if v x then l ++ [n] ++ r
                                                    else l ++ r) []
```

Řešení 9.1.1

a) `head [id, not]`

Nejprve určíme typy základních podvýrazů (musíme dát pozor, aby typové proměnné v různých podvýrazech byly různé):

```
id :: a -> a
not :: Bool -> Bool
head :: [b] -> b
```

Dále si všimneme, že `id` a `not` jsou oba prvky stejného seznamu, a tudíž musí mít stejný typ. Unifikujeme:
`a -> a` ∼ `Bool -> Bool`
Protože typ `Bool` je méně obecný, dostáváme `a = Bool`, a tedy `[id, not] :: [Bool -> Bool]` podle typu prvků seznamu (`Bool -> Bool`).

Dále aplikujeme funkci `head` na seznam, proto musíme unifikovat typ prvního parametru v typu `head` s typem seznamu:
`[b]` ∼ `[Bool -> Bool]` a z toho `b` ∼ `Bool -> Bool`

Při aplikaci funkce na jeden parametr dojde k odstranění typu prvního parametru z typu funkce a zároveň k dosazení za všechny typové proměnné v prvním parametru zmíněné, celkově tak dostáváme typ
`head [id, not] :: Bool -> Bool`
Vidíme, že výsledek je ve skutečnosti funkcí, a lze jej tedy dále aplikovat, například:
`head [id, not] True` $\rightsquigarrow^*$ `True`

b) `\f -> f 42`

Při typování funkcí (lambda i pojmenovaných) musíme nejprve odvodit typy parametrů a výrazů na levé straně podle vzorů, v nichž jsou použity. Následně se podíváme na pravou stranu definice, odvodíme návratový typ a při tom typicky zpřesníme typy parametrů podle jejich použití.

Pokud se na levé straně nachází proměnná, pak její typ je nějaká dosud nepoužitá polymorfní proměnná, proto odvodíme na začátku
`f :: a`

Nyní se díváme na výrazy na pravé straně a otypujeme nejprve `42 :: Num b => b`. Každá celočíselná konstanta je sama o sobě tohoto typu a ten se může zpřesnit podle místa použití.

Hodnota `f` je aplikovaná na jeden parametr, z čehož odvodíme, že se musí jednat o funkci, a tedy zkonkrétníme typ na `f :: c -> d` (dostáváme unifikaci `a` ∼ `c -> d`), což je nejobecnější možný typ (unární) funkce.

`f` je však aplikovaná na `42`, z čehož dostáváme unifikaci
`c` ∼ `Num b => b`
a tedy po dosazení zpět do typu pro `f` tento typ:
`f :: Num b => b -> d`
(dosadili jsme `b` za `c`, protože `b` má typový kontext, a tedy je méně obecné).

Typ výrazu `f 42` pak dostáváme odtržením typu prvního parametru z typu `f`:
`f 42 :: d`
Typ celé pravé strany je tedy `d`, což bude i návratový typ.

Funkce má jediný parametr `f :: Num b => b -> d`, celkově tedy dostáváme typ:
`(\f -> f 42) :: Num b => (b -> d) -> d`.

Poznámka: Typový kontext se musí objevit vždy na začátku funkce, proto všechny typové kontexty sloučíme a dáme na začátek.

c) `\t x -> x + x > t x`

Funkce má dva parametry. Jejich vzory jsou proměnné, tedy neomezují jejich typ, proto dostáváme:
`t :: a`
`x :: b`

Doplníme závorky podle priorit operátorů `(+), (>)`:
`\t x -> (x + x) > t x`

Na pravé straně máme použity následující základní výrazy:
`(+) :: Num c => c -> c -> c`
`(>) :: Ord d => d -> d -> Bool`

Z použití `x` jako parametru v `(+)` odvodíme
`b` ∼ `Num c => c`
a typ podvýrazu
`x + x :: Num c => c`

Vidíme, že `f` je funkce, tedy zkonkrétníme typ:
`a` ∼ `e -> f`, tedy `t :: e -> f`

Zároveň však vidíme, že první argument `t` je `x :: Num c => c`, a tedy zkonkrétníme typ dále se substitucí `e ~ Num c => c`:
`t :: Num c => c -> f`.

Nyní se zaměříme na parametry operátoru `(>)`. Levý parametr je `x + x :: Num c => c` a pravý `t x :: f` (z typu `t`). Z typu `(>)` dostáváme unifikace:
`Ord d => d ~ Num c => c`
`Ord d => d ~ f`
Dostáváme tedy `d ~ f ~ c`, ale musíme sloučit kontexty, proto:
`t :: (Ord d, Num d) => d -> d`
`x :: (Ord d, Num d) => d`

Typ celého výrazu na pravé straně je pak:
`(x + x) > f x :: Bool` (z návratového typu `(>)`).
Pro typy parametrů vezmeme nejkonkrétnější odvozené typy (ty co jsme viděli naposledy), celkově dostáváme typ:
`(\f x -> x + x > f x) :: (Ord d, Num d) => (d -> d) -> d -> Bool`

Poznámka: Typový kontext zde nelze zjednodušit, protože `Ord` není nadtřídou `Num` ani naopak.

d) `\xs -> filter (> 2) xs`

Začínáme s

```
xs :: a
filter :: (b -> Bool) -> [b] -> [b]
2 :: Num d => d
(>) :: Ord e => e -> e -> Bool
```

Částečnou aplikací `(>)` na 2 dostáváme `Num d => d ~ Ord e => e` a
`(> 2) :: (Ord d, Num d) => d -> Bool`.

Dále částečnou aplikací `filter` na `(> 2)` dostáváme unifikaci `b -> Bool ~ (Ord d, Num d) => d -> Bool`, a tedy `b ~ (Ord d, Num d) => d` a
`filter (> 2) :: (Ord d, Num d) => [d] -> [d]`.

Nyní aplikujeme tuto funkci na argument `xs`, kde dostaneme
`a ~ (Ord d, Num d) => [d]`
a celkový typ pravé strany je
`filter (> 2) xs :: (Ord d, Num d) => [d]`.

Jediným parametrem funkce je `xs :: (Ord d, Num d) => [d]`, celkem tedy dostáváme typ
`(\xs -> filter (> 2) xs) :: (Ord d, Num d) => [d] -> [d]`

e) `\f -> map f [1,2,3]`

```
1 :: Num a => a
[1,2,3] :: Num a => [a]
map :: (b -> c) -> [b] -> [c]
f :: d -- parametr
```

Z aplikace `map f` dostáváme `d ~ b -> c`, a tedy `f :: b -> c`, `map f :: [b] -> [c]`.

Tuto funkci dále aplikujeme na `[1,2,3]`: `[b] ~ Num a => [a]`,
`map f [1,2,3] :: [c]`

```
f :: Num a => a -> c
```

Celá funkce je tedy typu `(\f -> map f [1,2,3]) :: Num a => (a -> c) -> [c]`.

f)
```
foo f True  = map f [1,2,3]
foo f False = filter f [1,2,3]
```

Začneme s

```
f :: a        -- 1. parametr
True :: Bool  -- 2. parametr
False :: Bool -- dalsi vzor pro 2. parametr, typy unifikovatelne => OK

map :: (b -> c) -> [b] -> [c]
filter :: (d -> Bool) -> [d] -> [d]
[1,2,3] :: Num a => [a]
```

Dále pokračujeme obdobně jako v předchozím případě, ale pro každý vzor zvlášť:

```
-- z prvniho vzoru:
f :: Num a => a -> c
map f [1,2,3] :: [c]

-- z 2. vzoru:
f :: Num a => a -> Bool
filter f [1,2,3] :: Num a => [a]
```

Unifikujeme oba typy pro `f`:
`c` $\sim$ `Bool`
`f :: Num a => a -> Bool`
a tedy zpřesníme typ `map f [1,2,3] :: [Bool]`

Rovněž i návratová hodnota musí být vždy stejná: `[Bool]` $\sim$ `Num a => [a]`

Z čehož odvodíme, že funkce je otypovatelná pouze za předpokladu, že typ `Bool` je instancí typové třídy `Num`, což není pravda, a tedy funkci `foo` nelze otypovat.

g) `\(p,q) z -> q (tail z) : p (head z)`

```
p :: a
q :: b
(p,q) :: (a,b) -- prvni argument
z :: c         -- druhy argument
tail :: [d] -> [d]
head :: [e] -> e
(:) :: f -> [f] -> [f]
```

Z použití v `tail, head` můžeme odvodit, že `z` je seznam:
`c` $\sim$ `[d]` $\sim$ `[e]`, `z :: [d]`.

Dále tedy: `tail z :: [d]`, `head z :: d`.

Funkce `q` je tedy aplikovaná na parametr typu `[d]`, a proto její typ bude `q :: [d] -> g` (unifikace `b` $\sim$ `[d] -> g`).

Obdobně pro `p`: `p :: d -> h` (unifikace `a` $\sim$ `e -> h` $\sim$ `d -> h`).

Tedy `q (tail z) :: g`, `p (head z) :: h` jsou parametry `(:)`, a tedy dostáváme:
`f` $\sim$ `g`, `[f]` $\sim$ `h` $\Rightarrow$ `[f]` $\sim$ `h` $\sim$ `[g]` a po dosazení:
`q (tail z) :: g`
`p (head z) :: [g]`.

Celý výraz na pravé straně má tedy typ
`q (tail z) : p (head z) :: [g]`.

A celá funkce
`(\(p,q) z -> q (tail z) : p (head z)) :: (d -> [g], [d] -> g) -> [d] -> [g]`.

Řešení 9.1.2

a) V obou řádcích definice lze ze vzoru odvodit, že první parametr je typu `Foo a`. Typová proměnná `a` je tu proto, že zatím nevíme, jaké budou požadavky na typ hodnot uvnitř `Foo` (pozor, samotné `Foo` by bylo špatně, protože to je unární typový konstruktor a jako takový nemůže mít hodnoty).

   V prvním řádku je pak `xs :: [a]` (z definice `Bar`), a tedy návratová hodnota je `[a]`.

   V druhém řádku je `x :: a` (z definice `Baz`), návratová hodnota je pak `[a]`.

   V obou případech jsou návratové hodnoty stejné, tedy nemusíme provádět unifikaci a návratová hodnota celého výrazu je `[a]`.

   Celkově dostáváme: `getlist :: Foo a -> [a]`.

b) Na začátku odvodíme `foo :: a`. Díky použití `getlist` však můžeme zpřesnit na `foo :: Foo b` a určit `getlist foo :: [b]` (unifikace `a` $\sim$ `Foo b`).

   Dále potřebujeme znát typy dalších použitých funkcí a výrazů:

```
foldr :: (c -> d -> d) -> d -> [c] -> d
(+)   :: Num n => n -> n -> n
0     :: Num m => m
```

   Nyní postupně typujeme aplikaci `foldr`:

```
foldr (+) :: Num n => n -> [n] -> n       -- c ~ d ~ Num n => n
foldr (+) 0 :: Num n => [n] -> n          -- Num m => m ~ Num n => n
foldr (+) 0 (getlist foo) :: Num n => n   -- b ~ Num n => n
```

   Kombinací dostáváme typ lambda funkce:
   `(\foo -> foldr (+) 0 (getlist foo)) :: Num n => Foo n -> n`

Řešení 9.1.3

```
minmax :: Ord a => [a] -> (a, a)
minmax [x]    = (x, x)
minmax (x:xs) = let (mi, ma) = minmax xs in (min x mi, max x ma)

minmax' :: Ord a => [a] -> (a, a)
minmax' (x:xs) = foldr (\x (mi, ma) ->  (min x mi, max x ma)) (x, x) xs

minmaxBounded :: (Ord a, Bounded a) => [a] -> (a, a)
minmaxBounded [] = (maxBound, minBound) -- proc musi byt tohle naopak?
```

```
minmaxBounded (x:xs) = let (mi, ma) = minmaxBounded xs in (min x mi, max x
    ma)

minmaxBounded' :: (Ord a, Bounded a) => [a] -> (a, a)
minmaxBounded' = foldl (\ (mi, ma) x -> (min x mi, max x ma))
                       (maxBound, minBound)
```

Řešení 9.1.4

a)
```
\f -> (map . uncurry) f
\f -> map (uncurry f)
\f xs -> map (uncurry f) xs

\f xs -> [ f a b | (a, b) <- xs ]
```

b)
```
(\f xs -> zipWith (curry f) xs xs) :: ((a, a) -> b) -> [a] -> [b]
-- funkce je jiz v pointwise

\f xs -> [ f (x, x) | x <- xs ]
```

c)
```
\xs -> (map (* 2) . filter odd . map (* 3) . map (`div` 2)) xs
\xs -> map (* 2) (filter odd (map (* 3) (map (`div` 2) xs)))

\xs -> [ y * 2 | x <- xs, let y = div x 2 * 3, odd y ]
```

d)
```
\xs -> (map (\f -> f 5) . map (+)) xs
\xs -> map (\f -> f 5) (map (+) xs)

\xs -> [ (\f -> f 5) ((+) x) | x <- xs ]
\xs -> [ x + 5 | x <- xs ] -- zjednoduseni (aplikace lambda funkce)
```

Řešení 9.1.5

a)
```
readFile "/etc/passwd" :: IO String
putStrLn "bla" :: IO ()
```

Protože typ výsledku operátoru `(>>)` je stejný jako typ jeho druhého argumentu, typ celého výrazu je `IO ()`.

Přepis do `do`-notace:

```
do readFile "/etc/passwd"
   putStrLn "bla"
```

b)
```
f :: a
putStrLn "bla" :: IO ()
(>>=) :: IO b -> (b -> IO c) -> IO c

-- castecna aplikace (>>=)
(>>=) (putStrLn "bla") :: (() -> IO c) -> IO c

-- a tedy dostavame typ f:
f :: () -> IO c
```

```
-- a celeho vyrazu:
(\f -> putStrLn "bla" >>= f) :: (() -> IO c) -> IO c
```

Přepis do do-notace:

```
\f -> do x <- putStrLn "bla"
         f x
```

c) `getLine >>= \x -> return (read x)`

```
getLine :: IO String
return :: a -> IO a
read :: Read b => String -> b

-- z typu getLine a read:
x :: String

read x :: Read b => b
return (read x) :: Read b => IO b
getLine >>= \x -> return (read x) :: Read b => IO b
```

Přepis do do-notace:

```
do x <- getLine
   return (read x)
```

d) Nelze otypovat. Z `IO` není úniku, druhý parametr `(>>=)` musí vždy vracet `IO` akci (snaha o substituci `Read a => IO a` $\sim$ `Integer`).

Řešení 9.1.6

a) Nekorektní, funkci `max` možno aplikovat maximálně na dva argumenty.
b) Korektní, `(False < True) || True` $\leadsto$ `True || True` $\leadsto$ `True`. `Bool` je instancí typové třídy `Ord`, a tedy hodnoty tohoto typu lze porovnávat.
c) Nekorektní. Sice `5.1 ^ (3.2 ^ 2)` $\leadsto$ `5.1 ^ 10.24`, ale umocňování `(^)` je typu

   ```
   (^) :: (Num a, Integral b) => a -> b -> a
   ```

   a hodnotu `10.24` není možno otypovat `(Integral b) => b`.
   Poznámka: Šlo by použít jiný operátor umocnění. Haskell poskytuje tři a liší se povoleným typem argumentů:

   ```
   (^) :: (Integral b, Num a) => a -> b -> a
   (^^) :: (Fractional a, Integral b) => a -> b -> a
   (**) :: Floating a => a -> a -> a
   ```

d) Korektní, `2 ^ (if (even m) then 1 else m)`. Předpokládá se, že `m` je definováno (a je celočíselné).
e) Nekorektní. První chybou je chybný zápis funkce `mod`. Správně je buď `mod m 2` nebo `` m `mod` 2 ``. Další problém tvoří typy:

   ```
   mod :: (Integral a) => a -> a -> a,
   (/) :: (Fractional b) => b -> b -> b,
   (+) :: (Num c) => c -> c -> c
   ```

   Po aplikaci argumentů na funkce `mod` a `(/)` dostaneme

```
mod m 2 :: (Integral a) => a
11 / m :: (Fractional b) => b
(+) :: (Integral a, Fractional a) => a -> a -> a
```

`Integral` a `Fractional` jsou však nekompatibilní typové třídy, neexistuje tedy typ patřící do obou tříd. Otypování tedy není možné a výraz je nekorektní.
Tento příklad ilustruje fakt, že typový systém může odmítnout i výrazy, které jsou intuitivně korektní. V tomhle případe to můžeme vyřešit následovně:

```
fromIntegral (mod m 2) + 11 / m
```

f) Korektní, `((/) 3 2) + 2`.
g) Nekorektní. Uspořádané dvojice a trojice (ve všeobecnosti libovolné k-tice) jsou vždy navzájem různé typy. Seznam vytvořený v zadání by tedy nebyl homogenní.
h) Nekorektní, abstraktor `\s` je umístěný v prvním prvku uspořádané dvojice, tedy má platnost jenom tam. Výraz by však mohl být korektní, jestli by bylo `s` definováno na vyšší úrovni.
i) Korektní, definovanou lokální proměnnou `m` není nutné použít.
j) Nekorektní, v druhé členu trojice nesedí typy -- oba řetězce jsou stejného typu.
k) Korektnost závisí na typech `f` a `x`. Výraz je korektní, pouze když typ `f t` (kde `t` je prvek seznamu `x`) je kompatibilní s typem `[a]`.
l) Korektní, vytvoří jednoprvkový seznam.
m) Korektní, ekvivalentní prázdnému seznamu: `fst (map, 3) fst []` ⇝ `map fst []` ⇝ `[]`.
n) Nekorektní, zápis `[`a`,`b`..`c`]` je možné použít jenom u typů nacházejících se v typové třídě `Enum`.
o) Nekorektní, výraz `(x:_)` je vzor, který není možno použít na místech pro výraz. Vzory lze použít pouze v λ-abstrakci, jako argumenty při definici funkce a v konstrukcích `case`, `where` nebo nalevo od `<-`.
p) Nekorektní, funkce `fst` operuje pouze na uspořádaných dvojicích, ne na seznamech.
q) Korektní, v generátoru lze použít na levém místě vzor, tedy i `_`. V tomto případě je výsledkem prázdný seznam, protože generátor neposkytne žádný prvek.
r) Nekorektní, syntax intensionálních seznamů je `[ expr | rule, ..., rule ]`. Nemůžeme uvést svislítko a část s pravidly dvakrát.
s) Nekorektní. Zápis u druhého seznamu není možný. Před `..` lze uvést nejvíce dvě hodnoty -- počáteční a druhou (pro určení diference). Navíc před dvěma tečkami se nikdy neuvádí čárka.
t) Korektní. `getLine` vrací vnitřní hodnotu typu `String`, kterou pomocí operátoru `>>=` dáme jako argument funkci `putStrLn`.
u) Nekorektní. Vnitřní hodnotu `getLine` zahodíme, ale za `>>` musí být výraz typu `IO a`, avšak `putStrLn :: String -> IO ()`, a tedy tyto dva typy nejsou kompatibilní.
v) Korektní. Výsledek `getLine` se zahodí a vypíše se řetězec získaný z λ-abstrakce.
w) Korektní, jde o funkci, která bere jako vstup řetězcový argument a na výstup vypíše nejprve `OK` a následně zadaný řetězec. Následovné výrazy jsou totiž ekvivalentní:

```
(>>) (putStrLn "OK") . putStrLn
\x -> (>>) (putStrLn "OK") (putStrLn x)
putStrLn "OK" >> putStrLn x
```

x) Nekorektní, `f >>= g :: IO a`, avšak celý tento výraz je argumentem prvního `>>=` a musí mít tedy typ kompatibilní s typem `b -> IO c`, což neplatí.

Řešení 9.1.7

a) Nekorektní, správně je buď `(Int, Int)`, tj. uspořádaná dvojice tvořená dvěma hodnotami typu `Int`, anebo `[Int]`, tj. seznam hodnot typu `Int`.
b) Korektní, ekvivalentní s typem `Int`.
c) Nekorektní, zapsaný výraz je ekvivalentní s `[()] -> []`, avšak `[]` je pouze unární typový konstruktor (ne typ) a vždy musí „obalovat" nějaký typ.
d) Korektní.
e) Korektní.
f) Nekorektní, typové proměnné musí začínat malým písmenem, jinak jde o typový konstruktor, avšak `A` není standardním typovým konstruktorem.
g) Nekorektní, unární typový konstruktor `[]` nemá argument.
h) Korektní, `IO` je běžný unární typový konstruktor. Výrazem s kompatibilním typem je například `return putStrLn`.
i) Korektní, například `return (Just 1)` má kompatibilní typ.
j) Korektní.
k) Korektní, všechny použité názvy jsou typové proměnné (nemůžou to být obyčejné typy, jelikož začínají malým písmenem). Tento typ je ekvivalentní typu `[a] -> b -> c -> a`
l) Nekorektní, typový kontext musí odkazovat na typovou proměnnou, která je použitá. Tedy například když při určování typu výrazu vypadne typová proměnná, na kterou je navázána typová třída, je potřeba toto omezení z typového kontextu vypustit.
m) Nekorektní, typový kontext musí být vždy umístěn na začátku typu.
n) Korektní, sice každý typ v typové třídě `Integral` je i v typové třídě `Num`, a tedy uvedení `Num` je zbytečné, není to chybou. Jenom je potřeba myslet na to, že pokud typový kontext obsahuje více omezení, musí být umístěny v závorkách.
o) Nekorektní, typový kontext nelze takhle zkracovat. Musí být vždy uveden na začátku typu, tj. `Num a => a -> a`.
p) Nekorektní, typovým kontextem nelze nahradit typovou proměnnou, správně by mohlo být třeba `Num a => a -> c -> c`.

Řešení 9.1.8

a) `[2]`
b) Dojde k chybě vyhodnocování -- `head`, `tail` nejsou definovány na prázdném seznamu.
c) `[]`
d) `[]`
e) `[map (0:) []] ⇝ [[]]`
f) `[(++[]) [[], []], (++[]) [], (++[]) [[]]] ⇝`
   `⇝ [[[], []], [], [[]]]`
g) `[]`
h) `[[]]`
i) `[]`
j) `3 * 5 + (\x -> x + x ^ 2) (2 * 5 - 1) ⇝`
   `⇝ 15 + (2 * 5 - 1) + (2 * 5 - 1) ^ 2 ⇝* 15 + 9 + 9 ^ 2 ⇝ 105`
k) `(.) id (max 5) 3 ⇝ id (max 5 3) ⇝* 5`
l) `map f (x ++ []) ⇝ map f x`

Řešení 9.1.9

a) Ne, první výraz se snaží aplikovat funkci `(+1)` na `(*2)`, což není číslo. Druhý je ekvivalentní s výrazem `\x -> (+1) ((*2) x)`, a tedy s `\x -> x * 2 + 1`.
b) Ne, správně má být `f . (.g)` ⇝ `\x -> f ((.g) x)` ⇝ `\x -> f (x . g)`
c) Ne, tělo λ-abstrakcí se táhne tak daleko doprava, jak je to možné (v tomhle případě je to možné až po úplný konec výrazu). Implicitní uzávorkování je následovné:

```
getLine >>= (\x -> (putStrLn (reverse x)) >> (putStrLn "done"))
```

d) Ne, ve všeobecnosti není možné dělat „lifting“ λ-abstrakce tímto způsobem. První výraz byl nekorektní (definice `(+)` neumožňuje sečíst funkci a hodnotu), zatímco druhý výraz být korektní může.
e) Ne, operandy operátoru `&&` se vyhodnocují zleva. V případě, když `s1 == s2` a oba seznamy jsou nekonečné, vyhodnocování levého argumentu `&&` nikdy neskončí. Nedojde tedy ke zjednodušení celého výrazu na `False` i když druhým argumentem je `False`.

Řešení 9.1.10

a) `[a]`
b) `[()]`
c) `[Bool]`, tady pozor na to, že výsledkem je sice `[]` a ten má například po otypování v interpretu typ `[a]`, avšak tento prázdný seznam vznikl ze seznamu typu `[Bool]` a této stopy se už nelze zbavit.
d) `(a -> c) -> a -> c`
e) `b -> (b -> c) -> c`
f) `a -> a`
g) `(a -> b -> c) -> a -> (d -> b) -> d -> c`
h) `[a -> b -> c] -> [b -> a -> c]`
i) `(b -> b1 -> c) -> (a -> b) -> a -> (a1 -> b1) -> a1 -> c`
j) `((a -> [a] -> [a]) -> [a1] -> t) -> t`
k) `[[Bool]] -> [[Bool]]`
l) Nesprávně utvořený výraz -- nemožno sestrojit nekonečný typ.
m) `[t] -> [a]`
n) `IO a -> IO String`
o) `IO ()`, typem u `do`-konstrukcí je vždy typ posledního výrazu/akce.

Řešení 9.1.11 Funkce vybere nultý prvek, zahodí $k - 1$ prvků a rekurzivně se zavolá.

```
nth1 :: Int -> [a] -> [a]
nth1 _ []     = []
nth1 k (x:s) = x : nth1 k (drop (k - 1) s)
```

Funkce si udržuje index prvku i (modulo k) a jestli je rovný 0, prvek použije, jinak ho zahodí.

```
nth2 :: Int -> [a] -> [a]
nth2 k s = nth2' k 0 s where
    nth2' k i (x:s) = (if i==0 then (x:) else id) $ nth2' k (mod (i+1) k) s
    nth2' _ _ []    = []
```

Nejdříve „slepí“ seznam se seznamem `[0..]` a vybere pouze ty prvky, které jsou připojené k násobkům čísla k.

```
nth3 :: Int -> [a] -> [a]
nth3 k s = map fst $ filter ((==0) . flip mod k . snd) $ zip s [0..]
```

Vytvoří seznam seznamů po k prvcích a následně z nich vybere pouze první prvky.

```
nth4 :: Int -> [a] -> [a]
nth4 k s = map head $ nth4' k s where
    nth4' _ [] = []
    nth4' k s  = take k s : nth4' k (drop k s)
```

Řešení 9.1.12

```
modpwr :: Integer -> Integer -> Integer
modpwr _ 0 _ = 1
modpwr n k m = mod (if even k then t else n * t) m
    where t = modpwr (mod (n^2) m) (div k 2) m
```

Méně efektivní řešení s lineární časovou složitostí by mohlo vypadat takhle:

```
modpwr' :: Integer -> Integer -> Integer
modpwr' _ 0 _ = 1
modpwr' n k m = mod (n * modpwr' n (k-1) m) m
```

Řešení 9.1.13

a) Funkce `f1` bere jako argument funkci, která dostane argument 0.

```
f1 :: Num b => (b -> c) -> c
f1 x ⇝ flip id 0 x ⇝ id x 0 ⇝ x 0
```

b) Funkce `f2` vloží svůj argument do seznamu.

```
f2 :: a -> [a]
f2 x ⇝ flip (:) [] x ⇝ (:) x [] ≡ x:[] ≡ [x]
```

c) Funkce `f3` vezme dva seznamy a vrátí první seznam zkrácený na délku kratšího z nich.

```
f3 :: [a] -> [b] -> [a]
f3 [1,2,3] [4,5] ⇝ zipWith const [1,2,3] [4,5] ⇝* [const 1 4, const 2
   5] ⇝* [1,2]
```

d) Funkce `f4` vezme hodnotu typu `Bool` a vrací funkci, která pro `True` vrátí svůj argument s předřetězeným `'/'` a pro `False` vrátí původní argument beze změny.

```
f4 :: Bool -> [Char] -> [Char]
f4 True "yes" ⇝ (if True then ('/':) else id) "yes" ⇝ ('/':) "yes" ≡
   "/yes"
```

e) Funkce `f5` postupně odzadu aplikuje funkce ze seznamu v prvním argumentu na hodnotu v druhém argumentu.

```
f5 :: Num b => [b -> b] -> b
f5 [(3*),(+7)] ⇝ foldr id 0 [(3*),(+7)] ⇝* id (3*) (id (+7) 0) ⇝* (3*)
   ((+7) 0) ≡ 3 * (0 + 7)
```

f) Funkce `f6` je ekvivalentní s funkcí `foldr (\x s -> not s) True`. Hodnota prvků v seznamu se tedy vůbec nevyužívá, avšak za každý prvek se vygeneruje jedno `not` a všechny se postupně aplikují na `True`. Funkce tedy zjišťuje, jestli má seznam sudou délku.

```
f6 :: [a] -> Bool
f6 [1,2,3] ⇝* not (not (not True)) ⇝ False
```

Řešení 9.1.14

a) Platí, ale jenom jestli je `s` konečné.
b) Neplatí.
`map f . filter (p . f)` ≡ `filter p . map f`
c) Neplatí kvůli speciálnějšímu typu `flip . flip`. Pokud bychom nebrali typy do úvahy, platilo by.
`flip . flip` ≡ `(id :: (a -> b -> c) -> (a -> b -> c))`
d) Neplatí.
`foldr f (foldr f z s) t` ≡ `foldr f z (t ++ s)`
e) Neplatí, seznamy mohou být různé délky nebo některý může být nekonečný.
f) Neplatí pro prázdný seznam.
g) Platí.
h) Platí.

Řešení 9.1.15

a) `\x y -> f (g x) y` ≡ `f . g`
b) `\x y -> f (g (h1 x) (h2 y))`
c) `\x y -> if p y then f x y else x`
d) Doplnění není možno provést, protože čtvrtý argument funkce `foldr` není seznam, ale uspořádaná dvojice.
e) `\x y -> y` ≡ `flip const`

Řešení 9.1.16

a) Konstantní.
b) Lineární.
c) Lineární.
d) Konstantní.
e) Lineární k minimu hodnot `m`, `n`.
f) Lineární k délce seznamu `m`.
g) Výpočet nekončí.
h) Konstantní.
i) Konstantní.

Řešení 9.2.1

```
hanoi :: Int -> Int -> Int -> [(Int,Int)]
hanoi 1 source dest = [(source, dest)]
hanoi n source dest = (hanoi (n-1) source (6 - source - dest)) ++
                      (hanoi 1 source dest) ++
                      (hanoi (n-1) (6 - source - dest) dest)
```

Řešení 9.2.2 Budeme postupovat matematickou indukcí. Hledaným tvrzením je, že tyto funkce jsou ekvivalentní `($)`. Báze indukce je zřejmá. Nechť $\texttt{dollar}_n$ je funkce s n výskyty `($)`. Předpokládejme, že $\texttt{dollar}_n$ ≡ `($)`. Pak $\texttt{dollar}_{(n+1)}$ ≡ `($) .` $\texttt{dollar}_n$ ≡ `($) . ($)` a stačí tedy dokázat `($)` ≡ `($) . ($)`. Máme

`($) . ($)` ≡ `\x -> ($) (($) x)` ≡ `\x y -> (($) x) $ y` ≡ `\x y -> (($) x) y` ≡ `($)`.

Intuitivně, operátor `($)` funguje jako identita na unárních funkcích, a tedy složení identit je logicky opět identita.

Řešení 9.2.3

```
if' :: Bool -> a -> a -> a
if' cond th el = [el, th] !! fromEnum cond
```

nebo také

```
if' :: Bool -> a -> a -> a
if' True  t f = t
if' False t f = f
```

Řešení 9.2.4

```
head $ [ val1 | cond1 ] ++ [ val2 | cond2 ] ++ ... ++ [ val_default ]
```

Řešení 9.2.5

a) Lze nahlédnout, že ve výrazu `zipWith id x y` musí být `x` a `y` seznamy. Sémantiku lze nejsnáze přiblížit příkladem:

   ```
   zipWith id [f, g, h] [x, y, z] ⇝ [f x, g y, h z]
   ```

   První seznamový argument lze upravit následovně:

   ```
      map map [even, (/=0) . flip mod 7]
   ⇝ [map even, map ((/=0) . flip mod 7)]
   ```

   Protože tento seznam má dva prvky a v druhém seznamovém argumentu je použit `repeat`, výsledek aplikace `zipWith` bude mít vždy dva prvky. Tedy výraz ze zadání můžeme na základě těchto znalostí upravit na následovný ekvivalentní výraz:

   ```
   \s -> [map even s, map (/=0) . flip mod 7) s]
   ```

   Teď vidíme, že vzhledem na fakt `even :: Integral a => a -> Bool` a `mod :: Integral a => a -> a -> a` je typ našeho výrazu `Integral a => [a] -> [[Bool]]`. Slovně popsáno, náš výraz vrací pro seznam celých čísel seznam, kterého prvním prvkem je seznam indikující paritu vstupního seznamu a druhým prvkem je seznam indikující jestli dávají prvky vstupního seznamu nenulový zbytek po dělení sedmi:

   ```
   f [1..10] ⇝* [[False,True,False,True,False,True,False,True,False,True],
       [True,True,True,True,True,True,False,True,True,True]]
   ```

b) Výraz představuje funkci, která se v knihovně nazývá `swap`.

   ```
   uncurry (flip const) ≡ snd, uncurry const ≡ fst
   \t -> (snd t, fst t)
   \(a, b) -> (b, a)
   ```

c) Funkce `f` vrací faktoriál svého argumentu: První argument funkce `g` slouží jako akumulátor, do kterého se postupně vytváří složení funkcí `(n*)` pro všechny `n` mezi 1 a hodnotou argumentu funkce `f`. Na závěr se tato násobící funkce aplikuje na jedničku:

   ```
   f 0 ⇝ g id 0 ⇝ id 1 ⇝ 1
   f 1 ⇝ g id 1 ⇝ g (id . (1*)) 0 ⇝ (id . (*1)) 1 ⇝* 1 * 1
   f 2 ⇝ g id 2 ⇝ g (id . (2*)) 1 ⇝ g (id . (2*) . (1*)) 0 ⇝* 2 * 1 * 1
   f 3 ⇝ g id 3 ⇝ g (id . (3*)) 2 ⇝ g (id . (3*) . (2*)) 1 ⇝
   ```

`g (id . (3*) . (2*) . (1*)) 0` $\rightsquigarrow^*$ `3 * 2 * 1 * 1`

...

d) `\k -> concat . replicate k` ≡ `\k x -> concat (replicate k x)` Označme si první argument funkce `foldr` pro snazší manipulaci jako `corep`. Při vyhodnocování budeme pro lepší názornost postupovat striktní vyhodnocovací strategií, tj. od nejvnitřnějších volání funkcí. Pak dostáváme následovné výrazy:

`f 0` $\rightsquigarrow$ `[0]`
`f 1` $\rightsquigarrow$ `foldr corep [0] [1]` $\rightsquigarrow^*$ `[0]`
`f 2` $\rightsquigarrow$ `foldr corep [0] [2,1]` $\rightsquigarrow$ `corep 2 (foldr corep [0] [1])` $\rightsquigarrow^*$
`corep 2 [0]` $\rightsquigarrow^*$ `[0,0]`
`f 3` $\rightsquigarrow$ `foldr corep [0] [3,2,1]` $\rightsquigarrow$ `corep 3 (foldr corep [0,0] [2])` $\rightsquigarrow^*$
`corep 3 [0,0]` $\rightsquigarrow^*$ `[0,0,0,0,0,0]`
...

Na základě fungování funkce si lze všimnout, že `f` generuje pro argument `n` seznam s `n!` prvky `0`.

e) `skipping [1,2,3,4]` $\rightsquigarrow$ `[[2,3,4], [1,3,4], [1,2,4], [1,2,3]]`

f) Nejprve se podíváme na funkci `g`, jelikož nezávisí na `f`.

`g = [1,3..]`

Následně

`f = 0 : zipWith (+) f [1,3..]`
`f !! n == g !! (n - 1) + f !! (n - 1) == 2 * n - 1 + f !! (n - 1)`

Odsud se pomocí indukce dá dokázat `f == map (^2) [0..]`, tedy `f` je seznam druhých mocnin nezáporných celých čísel.

g) Prohledáváním BFS (Breadth-first search) generuje všechny konečné seznamy typu `[Bool]`:

`[[], [False], [True], [False, False], [True, False], ...]`

přičemž ke každému seznamu vytvoří dva nové tak, že na začátek jednoho připojí `False` a na začátek druhého `True`. Seznam generuje jako frontu.

h) Definice funkcí `f` a `g` jsou až na záměnu `f` a `g` totožné. Celý výraz tedy můžeme přepsat do tvaru `f = 1 : 1 : zipWith (tail f) f in f`, což představuje seznam Fibonacciho čísel.

Řešení 9.2.6

```
find l s = [p | p <- l, or (map (\x -> p == zipWith const x p) (suffixes
   s))]
  where
    suffixes "" = []
    suffixes (x:s) = (x:s) : suffixes s
```

Řešení 9.2.7

a) Nelze, funkce není otypovatelná. Argument `x` musí mít nějaký typ, řekněme `a`. Z pravé strany dostáváme specializaci `x` ≡ `a1 -> a2`. Funkci typu `a1 -> a2` však můžeme aplikovat pouze na argument typu `a1`. Avšak jejím argumentem je opět `x` s typem `a1 -> a2`. Dostáváme tedy `a1` ≡ `a1 -> a2` což představuje tzv. *nekonečný typ*. Tato funkce tedy není otypovatelná, potažmo jí nelze definovat.

b) Lze, výraz je možné přepsat do tvaru `\x y -> x - (\x -> x * x + 2) y`. Překrývání formálního argumentu `x` ve vnitřní λ-abstrakci je povolené -- hodnota `x` v součinu bude daná nejvnitřnější (nejbližší) λ-abstrakcí.

c) Lze, funkci je možné zapsat jako `f k = foldr (.) id (replicate k tail)` nebo taky `f = drop`

d) Nelze, funkce `f` musí mít jednoznačný typ. Pro funkci v zadání by platilo

```
f 1 :: [a] -> a
f 2 :: [[a]] -> a
f 3 :: [[[a]]] -> a
```

což jsou nekompatibilní typy.

Řešení 9.2.8

```
unused1 = [x, y]
unused2 = if True then x else y
unused3 1 = x; unused3 2 = y
unused4 = y `asTypeOf` x
```

Řešení 9.2.9 `f = _ (x:y:s) -> (x + y) : (x:y:s)`

Řešení 9.2.10 Parametr `p` představuje zastavovací podmínku, parametr `h` modifikaci prvků předávaných „na výstup“, parametr `t` modifikaci seznamu, na kterém se bude dále pracovat, a parametr `x` iniciální hodnotu řídící běh `unfold`. Celkový typ je následovný:

```
unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
```

a) `map f = unfold null (f . head) tail`

b) Nelze tak, aby funkce `unfold` byla nejvíce vnější funkcí.

```
filter p s = unfold null head (dropWhile (not . even) . tail)
     . dropWhile (not . even)
```

c) Není možno, výstupem `unfold` je vždy seznam.

d) `iterate = unfold (const False) id`

e) `repeat = unfold (const False) id id`

f) `replicate n x = ((==0) . fst) snd (\(n,x) -> (pred n,x)) (n,x)`

g)
```
take n x = unfold ((==0) . fst) (head . snd)
     (\(n,s) -> (n-1, tail s)) (n,x)
```

h) `list_id = unfold null head tail`

i) `enumFrom = unfold (const False) head succ`

j) `enumFromTo m n = unfold (>n) head succ m`

Řešení 9.2.11 Množina je tvořena funkcemi $dot_1, dot_2, \ldots, dot_9$ a platí $dot_{n+4} \equiv dot_n$ pro $n \geq 6$.

```
dot_1 = \a b c -> a (b c)
dot_2 = \a b c d -> a b (c d)
dot_3 = \a b c d -> a (b c d)
dot_4 = \a b c d e -> a b c (d e)
dot_5 = \a b c d -> a (b (c d))
```

```
dot_6 = \a b c d e -> a (b c) (d e)
dot_7 = \a b c d e -> a b (c d e)
dot_8 = \a b c d e -> a (b c d e)
dot_9 = \a b c d e f -> a b c d (e f)
```

Řešení 9.2.12 Pouze pomocí `id` to možné není. Každý výskyt `id` se může přepsat podle definice na hodnotu jejího argumentu a jelikož jediná použitá hodnota je `id`, výsledkem je vždy funkce `id`.

Pomocí funkce `const` to však je možné udělat. Například funkce

```
const
const const
const (const const)
const (const (const const))
...
```

jsou ekvivalentní funkcím

```
\x1 x2 -> x1
\x1 x2 x3 -> x2
\x1 x2 x3 x4 -> x3
\x1 x2 x3 x4 x5 -> x4
...
```

což jsou zjevně navzájem různé funkce a je jich nekonečně mnoho.

Řešení 9.2.13 Uvažujme, že řešením jsou seznamy `s` typu `s :: [t]`. Nechť `|t|` představuje počet plně definovaných hodnot typu `t`. Rozeberme možnosti:

- `|t|` $= 0$
  Vzhledem na to, že neexistuje žádná plně definovaná hodnota tohoto typu, jediným seznamem, který za těchto podmínek bereme do úvahy, je `[]` a pro něj platí zadaná podmínka triviálně.
- `|t|` $= 1$
  Nutně `f = id`, odkud `map f = map id = id`, tedy podmínka platí po každý seznam typu `[t]`.
- `|t|` > 1 Rozeberme několik případů podle obsahu seznamu `s`:
  - `s = []`
    Podmínka platí triviálně.
  - `s` obsahuje pouze prvky `a :: t`
    Nechť `b` je hodnota typu `t`, jiná než `a`. Uvažme dále funkce `f = const b` a `p = (b /=)`. Pro tento výběr podmínka neplatí, protože `filter p (map f s) = []` $\not\equiv$ `map f (filter p s) = map f s`.
  - `s` obsahuje alespoň dva různé prvky `a, b :: t`
    Uvažme funkce `f = const a` a `p = (a/=)`. Pro tento výběr podmínka neplatí, protože `filter p (map f s) = []` $\not\equiv$ `map f (filter p s)`

Závěr: Dané tvrzení platí pouze pro prázdné seznamy libovolného typu a pro libovolné seznamy typu `[()]` nebo typu izomorfního.

Řešení 10.1.1 *SWI-Prolog* spustíme jednoduchým voláním `swipl` z příkazové řádky. Základní predikáty pro práci s interpretrem jsou uvedeny níže.

- **`help/0`** Zobrazí základní nápovědu o použití nápovědy.
- **`help/1`** Zobrazí nápovědu k predikátu v argumentu.
- **`apropos/1`** Zobrazí predikáty, které mají v názvu nebo popisu zadaný výraz.
- **`consult/1`** Načte (zkompiluje) zdrojový kód ze zadaného souboru.
- **`make/0`** Znovu načte zdrojový kód ze změněných souborů.
- **`halt/0`** Ukončí interpret *SWI-Prolog*.
- **`trace/0`** Zapíná ladící mód a krokování výpočtu (existuje i predikát **`notrace/0`**, který vypíná krokování, ale nechává ladící mód).
- **`nodebug/0`** Vypíná krokování i ladící mód (existuje i predikát **`debug/0`**, který zapíná ladící mód, ale nezapíná krokování výpočtu).

Mějte na paměti, že příkazy v Prologu musí být ukončeny tečkou, jinak interpret očekává, že příkaz pokračuje.

Řešení 10.1.2

a) `?- parent(petr,lenka).`
`true.` Ano, je.

b) `?- parent(petr,jan).`
`false.` Ne, není

c) `?- parent(pavla,X).`
`X = petr; X = tomas.` Pavla má děti Petra a Tomáše.

d) `?- parent(petr,X), woman(X).`
`X = lenka.` Petr má jednu dceru, Lenku.

e) `?- father(X,petr).`
`X = adam; false.` Otcem Petra je Adam.

f) `?- father(X,Y), man(Y).`
```
X = petr, Y = filip; X = pavel, Y = jan; X = adam, Y = petr;
X = tomas, Y = michal; X = michal, Y = radek; false.
```

Řešení 10.1.3

a) `child(Child, Parent) :- parent(Parent, Child).`

b) `grandmother(GM, GCH) :- woman(GM), parent(GM,CH), parent(CH,GCH).`

c)
```
stepBrother(Brother, Sibling) :- man(Brother),
  parent(Parent1, Brother), parent(Parent2, Brother),
  Parent1 \= Parent2, parent(Parent1, Sibling),
  Brother \= Sibling,
  parent(Parent3, Sibling), Parent3 \= Parent1, Parent3 \= Parent2.
```

Řešení 10.1.4

```
descendant(Desc,Anc) :- parent(Anc,Desc).
descendant(Desc,Anc) :- parent(Anc,Int), descendant(Desc,Int).
```

Dotaz na potomky Pavly vrátí osoby v následujícím pořadí: Petr, Tomáš, Filip, Lenka, Věra, Michal, Radek. Pořadí klauzulí v predikátu ovlivňuje způsob prohledávání stromu řešení. Kdybychom klauzule výše přehodili, našli bychom vzdálenější potomky (Věra, Radek) dříve než

jejich korespondující rodiče (Lenka, Michal). Pokud bychom přehodili cíle ve druhé klauzuli, způsobíme zacyklení výpočtu, neboť interpret se bude snažit nalézt nejdříve ty vzdálenější a vzdálenější potomky.

Řešení 10.1.5

```
daughter(marge, lisa).
daughter(homer, lisa).
daughter(marge, maggie).
daughter(homer, maggie).
son(marge, bart).
son(homer, bart).
```

Předpokládejme, že sestra je taková dcera, která se svým sourozencem sdílí alespoň jednoho rodiče. Implementujme predikát `sister/2` následovně:

```
sister(Person, Sister) :- daughter(Parent, Sister),
    daughter(Parent, Person), Sister \= Person.
sister(Person, Sister) :- daughter(Parent, Sister), son(Parent, Person).
```

Výše uvedené řešení vrátí na dotaz `?- sister(bart, X).` všechny Bartovy sestry, avšak každou dvakrát (každá je totiž sestrou i přes matku i přes otce). Vypisovat každou sestru pouze jednou by bylo poněkud složitější, bylo by potřeba podrobně rozepsat všechny možné případy.

Sestry Lisy zjistíme dotazem `?- sister(lisa, X).` Poznamenejme ještě, že vypsání Lisy jako své vlastní sestry zabraňuje podmínka neunifikovatelnosti na konci první klauzule řešení.

Řešení 10.2.1

a) Neuspěje, protože `a/1` a `b/1` jsou různé predikáty (mají různý název).
b) Uspěje, výsledná substituce [`X = a(Y)`].
c) Neuspěje, protože `a/1` a `a/2` jsou různé predikáty (mají různou aritu).
d) Uspěje se substitucí [`X = a(X)`]. Substituce je však nekorektní, protože vznikne nekonečný term. Prolog však kontrolu výskytu nedělá (je to náročnější problém, jak se zdá).
e) Uspěje, výsledná substituce [`X = Petr, Petr = plus`].
f) Uspěje, výsledná substituce [`X = 2, Y = 1, Z = q(w)`].
g) Uspěje se substitucí [`X = Z, Z = q(Z), W = 1`]. Substituce je však nekorektní, protože vznikne nekonečný term.
h) Uspěje se substitucí [`X = Y, Y = P, P = R, R = q(R), Z = 1`]. Substituce je však nekorektní, protože vznikne nekonečný term.

Řešení 10.2.2

a) Uspěje, výsledná substituce je prázdná.
b) Neuspěje, termy mají různá jména.
c) Uspěje, výsledná substituce je prázdná.
d) Uspěje, výsledná substituce [`Monday = monday`].
e) Neuspěje, termy mají různá jména.
f) Neuspěje, termy mají různá jména a arity.
g) Uspěje, výsledná substituce [`X = day(monday)`].
h) Uspěje, výsledná substituce [`X = monday`].
i) Uspěje, výsledná substituce [`X = tuesday, Y = monday`].

j) Neuspěje, unifikace selhala na podmínce `tuesday = wednesday`.
k) Neuspěje, termy mají různé arity.
l) Uspěje s výslednou substitucí [`day(D) = D`]. Substituce je však nekorektní, protože vznikne nekonečný term (Prolog kontrolu výskytu nedělá).
m) Uspěje, výsledná substituce [`X = day(saturday), Y = day(sunday)`].
n) Neuspěje, unifikace selhala na podmínce `saturday = sunday`.

Řešení 10.2.3

a) `X = f(Y).`
b) Výrazy nejsou unifikovatelné (unifikace v interpretru sice uspěje, ale není korektní).
c) `X = Y, Y = Z.`
d) `X = Y, Y = 8.`
e) Výrazy nejsou unifikovatelné, predikáty na nejvyšší úrovni mají různou aritu.
f) `X = Y, Y = f(Z).`
g) `X = W, W = f(g(Z)), Y = g(Z).`

Řešení 10.2.4

```
crossword( V1, V2, V3, H1, H2, H3 ) :-
    word( V1, _V11, V12, _V13, V14, _V15, V16, _V17 ),
    word( V2, _V21, V22, _V23, V24, _V25, V26, _V27 ),
    word( V3, _V31, V32, _V33, V34, _V35, V36, _V37 ),
    word( H1, _H11, H12, _H13, H14, _H15, H16, _H17 ),
    word( H2, _H21, H22, _H23, H24, _H25, H26, _H27 ),
    word( H3, _H31, H32, _H33, H34, _H35, H36, _H37 ),
    V12 = H12, V14 = H22, V16 = H32,
    V22 = H14, V24 = H24, V26 = H34,
    V32 = H16, V34 = H26, V36 = H36.
```

Řešení se dá napsat i trochu efektivněji, když namísto explicitních unifikací použijeme stejné proměnné už v predikátech pro slova:

```
crossword2( V1, V2, V3, H1, H2, H3 ) :-
    word( V1, _V11, V12, _V13, V14, _V15, V16, _V17 ),
    word( V2, _V21, V22, _V23, V24, _V25, V26, _V27 ),
    word( V3, _V31, V32, _V33, V34, _V35, V36, _V37 ),
    word( H1, _H11, V12, _H13, V22, _H15, V32, _H17 ),
    word( H2, _H21, V14, _H23, V24, _H25, V34, _H27 ),
    word( H3, _H31, V16, _H33, V26, _H35, V36, _H37 ).
```

Řešení 10.2.5

```
geq(_,0).
geq(s(X),s(Y)) :- geq(X,Y).
```

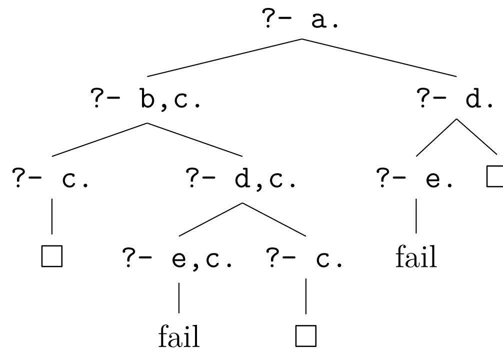
Řešení 10.2.6

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Řešení 10.2.7

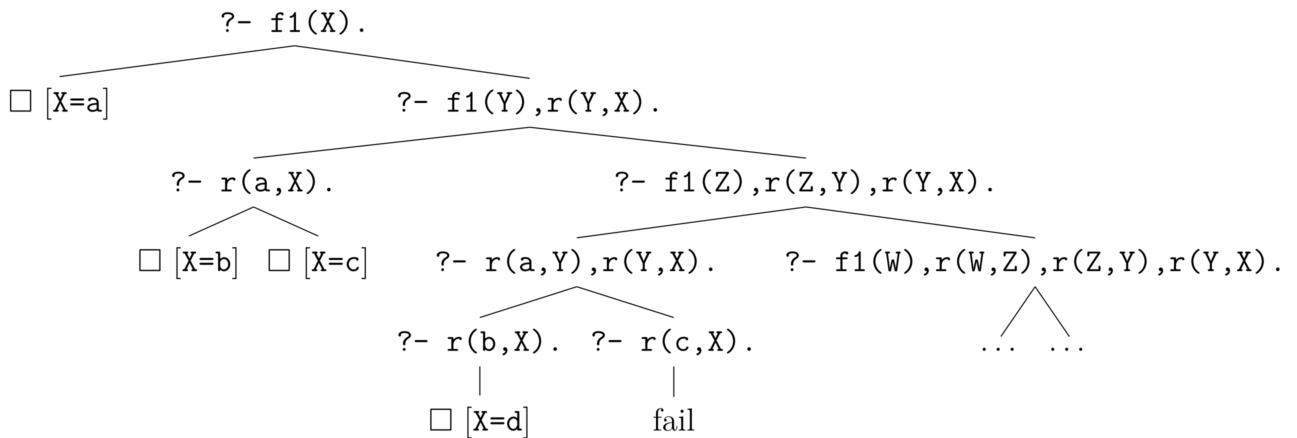
```
mult(_X,0,0).
mult(X,s(Y),Z) :- mult(X,Y,Z1), add(X,Z1,Z).
```

Řešení 10.3.1



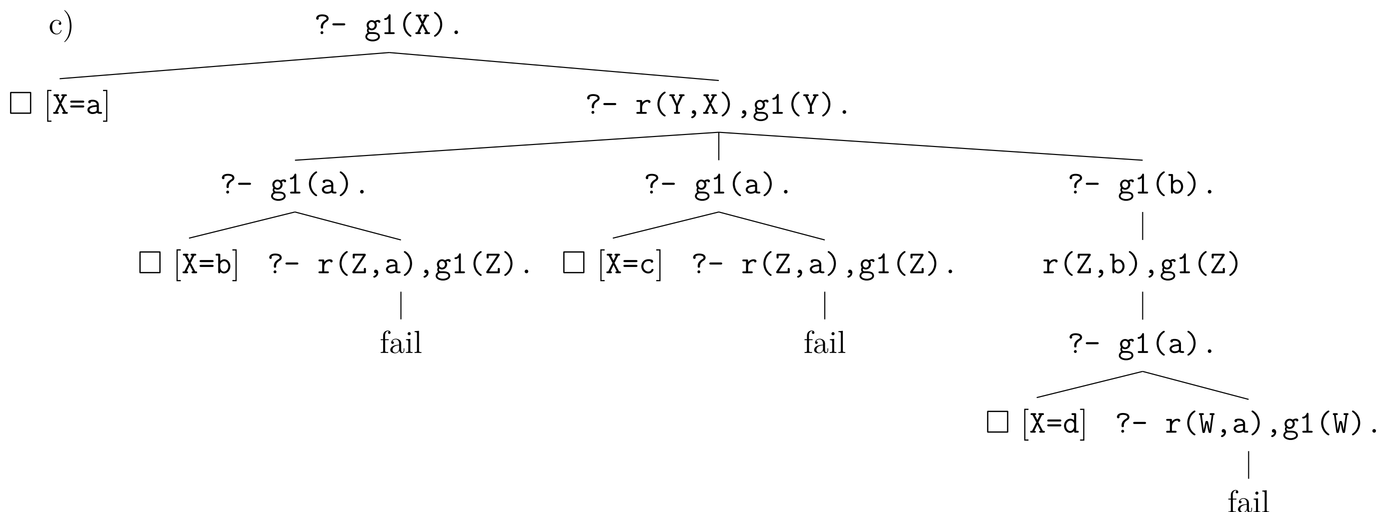
Řešení 10.3.2

a)



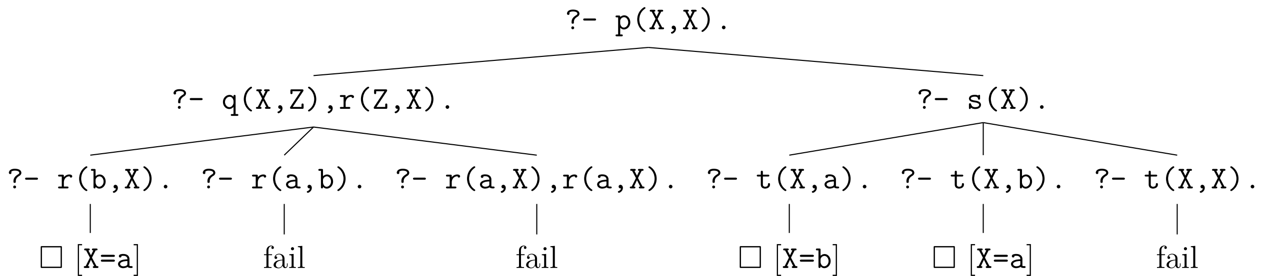
b) Výsledný SLD strom bude stejný jako v případě a), jenom každý podstrom, který má v kořeni výraz začínající predikátem `f1/1` bude zrcadlově převrácený. To však znamená, že nejlevější větev bude nekonečná, a tedy interpret bude cyklit.

c)



d) Výsledný SLD strom bude stejný jako v případě c), jenom každý podstrom, který má v kořeni výraz začínající predikátem `g1/1` bude zrcadlově převrácený. To znamená, že uvedená řešení budou nalezena v pořadí `X=b`, `X=c`, `X=d`, `X=a`.

Řešení 10.3.3



Řešení 11.1.1

a) Unifikace neuspěje, term `2` není unifikovatelný s termem `+(1, 1)`.
b) Unifikace uspěje.
c) Uspěje, aritmeticky se vyhodnotí pravý argument operátoru `is`, a poté unifikuje s levou stranou (tedy unifikujeme `2 = 2`).
d) Neuspěje, pravá strana se vyhodnotí na `2`, levá strana je ale term `1 + 1`, který není unifikovatelný s `2`.
e) Uspěje s přiřazením `X = 1 + 1`, unifikace nevyhodnocuje aritmetiku.
f) Uspěje s přiřazením `X = 2` díky aritmetickému vyhodnocení přes `is`.
g) Neuspěje, protože pravý argument `is` obsahuje volnou (nepřiřazenou) proměnnou.
h) Uspěje, do `X` je přiřazeno díky předchozí unifikaci.

Řešení 11.1.2

a) Uspěje, dojde k substituci.
b) Neuspěje, `X` není identické s `2`.
c) Uspěje, v okamžiku testování identity je `X` již nastaveno na `2`.
d) Neuspěje, `=:=` aritmeticky vyhodnocuje obě strany, a proto ani jedna nesmí obsahovat neinstanciovanou proměnnou.
e) Neuspěje, termy nejsou identické.
f) Uspěje, protože se nejprve aritmeticky vyhodnotí obě strany na `2`.
g) Uspěje, protože se nejprve aritmeticky vyhodnotí obě strany na `2`.

Řešení 11.1.3

a) Uspěje, obě strany se nejprve aritmeticky vyhodnotí a nerovnost platí.
b) Uspěje.
c) Uspěje.
d) Uspěje.
e) Uspěje, termy nejsou identické.
f) Neuspěje, obě strany se nejprve aritmeticky vyhodnotí, výsledné termy jsou ale identické.
g) Uspěje, obě strany se nejprve aritmeticky vyhodnotí a výsledné termy nejsou identické.

Řešení 11.1.4

a) Unifikace, uspěje se substitucí [`X = Y + 1`].
b) Nekorektní výraz, pravá strana operátoru `is` obsahuje proměnnou.
c) Unifikace, uspěje se substitucí [`X = Y`].
d) Porovnání na identitu, neuspěje (proměnné jsou různé).
e) Unifikace, neuspěje (různé termy).
f) Unifikace, neuspěje (různé termy).
g) Unifikace, uspěje s prázdnou substitucí.
h) Aritmetické vyhodnocení + unifikace, uspěje (vhodnější by však pravděpodobně bylo použít operátor `=:=`).
i) Aritmetické vyhodnocení + unifikace, unifikace neuspěje (ekvivalentní `1 + 1 = 2`, kde se liší term na nejvyšší úrovni -- `+/2` vs. `2/0`).
j) Aritmetické porovnání, uspěje.
k) Porovnání na (identickou) různost, uspěje.
l) Nekorektní aritmetické porovnání, výrazy na obou stranách nejsou instanciovány.
m) Aritmetická nerovnost, uspěje.
n) Nekorektní výraz, operátor `<=` neexistuje, správně je `=<`.
o) Aritmetické porovnání, uspěje.
p) V daném kontextu funguje podobně jako aritmetické porovnání, tedy neuspěje (`sin(2)` se vyhodnotí, ale `sin(X)` zůstává jako aplikace termu na proměnnou).
q) Unifikace, uspěje se substitucí [`X = 2 + Y`].
r) Nekorektní aritmetické porovnání, výrazy na obou stranách nejsou plně instanciovány.

Řešení 11.1.5

a) Uspěje (na porovnání na aritmetickou rovnost by se však měl použít operátor `=:=`).
b) Uspěje.
c) Unifikace neuspěje, termy jsou různé.
d) Neuspěje, termy nejsou identické.
e) Neuspěje, aritmetická nerovnost není splněna.
f) Uspěje, aritmetická rovnost platí.
g) Uspěje, termy jsou identické.
h) Neuspěje, termy nejsou identické.
i) Uspěje, aritmetická rovnost je splněna.
j) Neuspěje, aritmetická nerovnost není splněna.

Řešení 11.1.6

```
convert(0, 0).
convert(X, s(Y)) :- X1 is X - 1, convert(X1, Y).
```

Řešení 11.1.7

```
firstnums(N, S) :- firstnums(N, 0, S).
firstnums(0, S, S).
firstnums(N, X, S) :- N1 is N - 1, X1 is X + N, firstnums(N1, X1, S).
```

Toto řešení má nevýhodu v tom, že po prvně nalezeném řešení se Prolog pokusí nalézt další (v místě, kde bylo použito faktu, se pokusí použít pravidla), avšak žádné další nenalezne a výpočet neskončí. To lze napravit pomocí tzv. *řezů*, které budou probírány později. Fakt by se

upravil na `firstnums(0, S, S) :- !.` Řez (`!`) zakáže hledání dalšího řešení, pokud uspěje tento (původně) fakt.

Připomeňme také, že predikát v Prologu je definován nejen identifikátorem, ale také aritou, takže dvou- a tříargumentové `firstnums` jsou rozdílné a nezávislé predikáty.

Výše uvedené řešení sice funguje, avšak níže uvedené je efektivnější (využívá vzorec pro součet aritmetické posloupnosti)

```
firstnums2(N, S) :- S is (N * (N + 1)) // 2.
```

Použitý operátor `//` realizuje celočíselné dělení.

Řešení 11.1.8

```
fact(0, 1).
fact(N, Fact) :- N >= 0, M is N - 1, fact(M, FactP), Fact is N * FactP.
```

Řešení 11.1.9

```
powertwo(1).
powertwo(X) :- X mod 2 =:= 0, X1 is X // 2, powertwo(X1).
```

Řešení 11.1.10

```
prime(Number) :- testPrime(2, Number).

testPrime(P, P) :- !.
testPrime(D, P) :- D < P, P mod D =\= 0, D1 is D + 1, testPrime(D1, P).
```

Opět, v řešení používáme řez, abychom zamezili hledání dalšího řešení, které by neexistovalo.

Řešení 11.1.11

```
gcd(X, X, X).
gcd(A, B, X) :- A > B, A1 is A - B, gcd(A1, B, X).
gcd(A, B, X) :- B > A, B1 is B - A, gcd(A, B1, X).
```

Dodejme, že existuje i efektivnější verze využívající zbytky po dělení.

Řešení 11.1.12

```
dsum(N, Sum) :- dsum(N, 0, Sum).
dsum(0, Sum, Sum) :- !.
dsum(N, X, Sum) :- X1 is X + N mod 10, N1 is N // 10, dsum(N1, X1, Sum).
```

Opět, v řešení používáme řez, abychom zamezili hledání dalšího řešení, které by neexistovalo.

Řešení 11.1.13 Program je velice neefektivní (má exponenciální složitost). Hodnoty předchozích členů posloupnosti se počítají opakovaně. Nejlépe to uvidíte, když si nakreslíte strom výpočtu pro nějaké malé `N`, například `6`.

Takto zapsaný program kromě toho neumožňuje optimalizaci koncového volání (koncovou rekurzi nebo tail recursion).

Řešení 11.2.1

a) Korektní seznam s délkou 4.
b) Korektní seznam s délkou 3.
c) Nekorektní výraz (zbytek seznamu za znakem | není zapsaný správně).
d) Korektní seznam s délkou 4.
e) Korektní seznam s délkou 4.
f) Korektní seznam s délkou 1.
g) Nekorektní výraz (zbytek seznamu za znakem | není zapsaný správně).
h) Korektní seznam s délkou 5.

Řešení 11.2.2

```
head([H|_], H).

tail([_|T], T).

last([Last], Last).
last([_|T], Last) :- last(T, Last).

init([_], []).
init([H|Tail], [H|Init]) :- init(Tail, Init).
```

Řešení 11.2.3

a) `prefix(Prefix, List) :- append(Prefix, _, List).`

b) `suffix(Suffix, List) :- append(_, Suffix, List).`

c) `element(X, List) :- append(_Prefix, [X|_Suffix], List).`

d) `adjacent(X, Y, List) :- append(_Prefix, [X,Y|_Suffix], List).`

e)
```
sublist(Sub, List) :- append(_Prefix, SubSuffix, List),
    append(Sub, _Suffix, SubSuffix).
```

Řešení 11.2.4

```
map([], []).
map([H1|T1], [H2|T2]) :- mf(H1, H2), map(T1, T2).
```

Řešení 11.2.5 Predikát `something/2` odstraní ze seznamu v prvním argumentu duplicitní prvky (ponechá jenom první výskyt) a výsledný seznam unifikuje do druhého argumentu.

Řešení 11.2.6 V řešení využíváme knihovní funkci `member/2`, která uspěje, pokud je první argument prvkem seznamu v druhém argumentu.

a)
```
variation3(List, Variation3) :-
    member(A, List), member(B, List), A \= B,
    member(C, List), A \= C, B \= C, Variation3 = [A,B,C].
```

b)
```
combination3(List, Combination3) :-
    variation3(List, [X,Y,Z]), X @< Y, Y @< Z,
    Combination3 = [X,Y,Z].
```

Řešení 11.2.7

```
doubles([], []).
doubles([X1|T1], [X2|T2]) :- X2 is 2 * X1, doubles(T1, T2).
```

Řešení 11.2.8

a)
```
listLength([], 0).
listLength([_|T], Length) :-
    listLength(T, LengthTail), Length is LengthTail + 1.
```

b)
```
listSum([], 0).
listSum([H|T], Sum) :- listSum(T, SumTail), Sum is SumTail + H.
```

c)
```
fact(0, 1) :- !.
fact(N, Fact) :- NN is N - 1, fact(NN, SubFact), Fact is SubFact * N.
```

d)
```
scalar([], [], 0).
scalar([H1|T1], [H2|T2], Scalar) :-
    scalar(T1, T2, ScalarTail), Scalar is ScalarTail + H1 * H2.
```

Všechny predikáty se však dají naprogramovat i efektivněji: tak, aby se při výpočtu mohla použít optimalizace posledního volání. Myšlenkou je použít pomocnou proměnnou (akumulátor), ve kterém si budeme „střádat“ mezivýsledek. Následují všechna řešení přepsaná tímto způsobem. Pro pomocný predikát využívající akumulátor používáme stejné jméno, protože se liší v aritě.

a)
```
listLength2(List, Length) :- listLength2(List, Length, 0).
listLength2([], Length, Length).
listLength2([_|T], Length, Acc) :- Acc2 is Acc + 1,
                                   listLength2(T, Length, Acc2).
```

b)
```
listSum2(List, Sum) :- listSum2(List, Sum, 0).
listSum2([], Sum, Sum).
listSum2([H|T], Sum, Acc) :- Acc2 is Acc + H, listSum2(T, Sum, Acc2).
```

c)
```
fact2(N, Fact) :- fact2(N, Fact, 1).
fact2(0, Fact, Fact) :- !.
fact2(N, Fact, Acc) :- Acc2 is Acc * N, NN is N - 1,
    fact2(NN, Fact, Acc2).
```

d)
```
scalar2(List1, List2, Scalar) :- scalar2(List1, List2, Scalar, 0).
scalar2([], [], Scalar, Scalar).
scalar2([H1|T1], [H2|T2], Scalar, Acc) :- Acc2 is Acc + H1 * H2,
    scalar2(T1, T2, Scalar, Acc2).
```

Řešení 11.2.9

```
filter([], []).
filter([X|T1], [X|T2]) :- number(X), !, filter(T1, T2).
filter([_X|T1], T2) :- filter(T1, T2).
```

Řešení 11.2.10

```
digits(S, X) :- digits(S, 0, X).
digits([], X, X).
digits([A|T], P, X) :- P1 is 10 * P + A, digits(T, P1, X).
```

Řešení 11.2.11

```
nth(1, [X|_], X).
nth(N, [_|XS], X) :- N > 1, N1 is N - 1, nth(N1, XS, X).
```

Řešení 11.2.12

```
listSum(List, Sum) :- listSum2(List, 0, Sum).
listSum2([], Sum, Sum).
listSum2([X|XS], Acc, Sum) :-
    number(X), Acc1 is Acc + X, listSum2(XS, Acc1, Sum).

fact(N, F) :- fact2(N, 1, F).
fact2(0, F, F).
fact2(N, Acc, F) :-
    N > 0, Acc1 is N * Acc, N1 is N - 1, fact2(N1, Acc1, F).

fib(N, Fib) :- fib2(N, 0, 1, Fib).
fib2(0, Fib, _, Fib).
fib2(N, A, B, Fib) :-
    N > 0, B1 is A + B, N1 is N - 1, fib2(N1, B, B1, Fib).
```

Řešení 11.2.13

```
mean(S, X) :- mean(S, 0, 0, X).
mean([], Sum, Count, X) :- X is Sum / Count.
mean([H|T], Sum, Count, X) :-
    Sum1 is Sum + H, Count1 is Count + 1, mean(T, Sum1, Count1, X).
```

Řešení 11.2.14

```
zip([H1|T1], [H2|T2], [H1-H2|T]) :- zip(T1, T2, T).
zip([], [_|_], []).
zip(_, [], []).
```

Řešení 12.1.1

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.

% efektivnejsi reseni pomoci rezu
max2(X, Y, Z) :- X >= Y, !, Z = X.
max2(X, Y, Y).
```

Následující definice není korektní:

```
max(X, Y, X) :- X >= Y, !.
max(_X, Y, Y).
```

Dotaz `?- max(2, 1, 1).` uspěje. Nedá se totiž unifikovat s hlavou první klauzule, tudíž se interpretr ani nepokusí o vyhodnocení její pravé strany a rovnou přejde na druhou klauzuli. Ta uspěje, protože její platnost není vůbec závislá na hodnotě `_X`.

Pro přehlednost pojmenujme třetí argument predikátu `Z`. Problém je v tom, že první klauzule tvrdí `X = Z` ∧ `X >= Y` ⇒ `true`, zatímco správná definice je `X >= Y` ⇒ `Z = X`.

Naopak u `max2` výše tento problém není: nyní již není možné dostat se na druhou klauzuli v případě, že platí `X >= Y`, protože hlava první klauzule je vždy unifikovatelná s dotazem (a pokud `X >= Y` uspěje, řez zajistí, že se již nemůžeme dostat na druhou klauzuli).

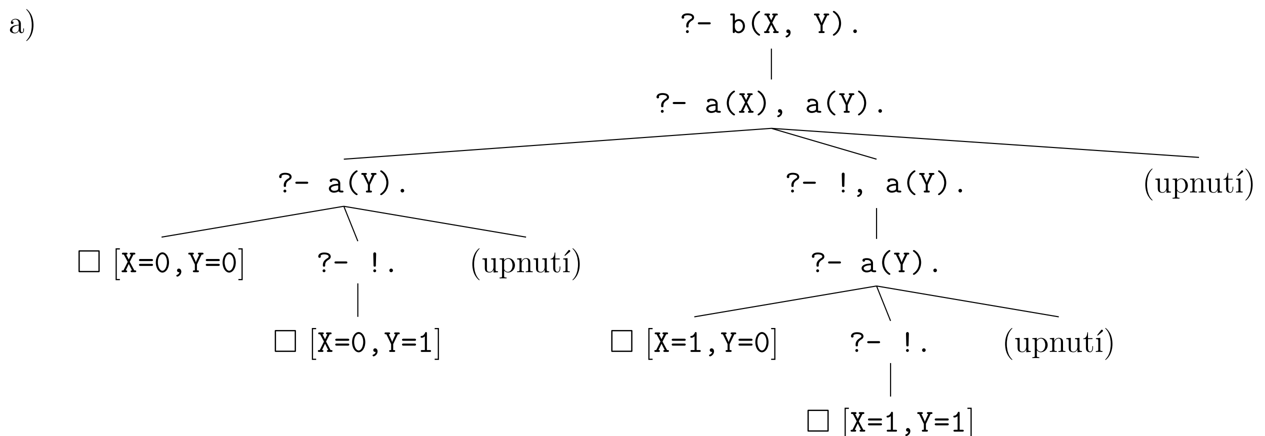
Řešení 12.1.2 Predikát `mem1/2` vyhledá všechny výskyty prvku. Při porovnávání hledaného prvku s prvky seznamu může dojít k vázání proměnných (může sloužit ke generování všech prvků seznamu).

Predikát `mem2/2` najde jenom první výskyt, také váže proměnné.

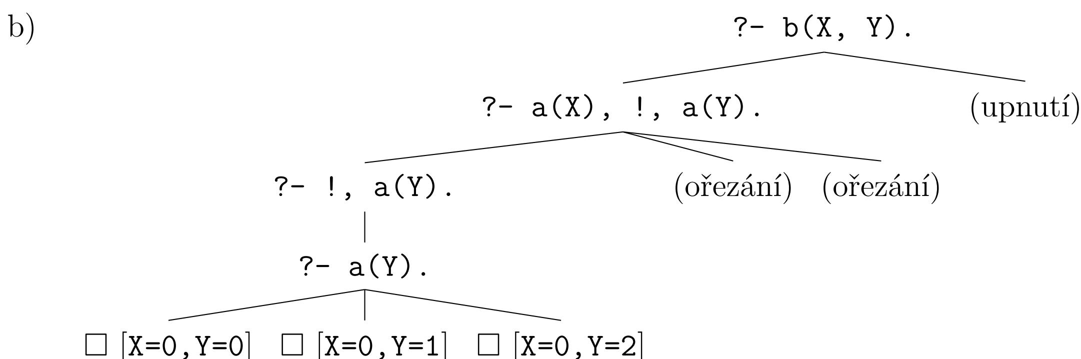
Predikát `mem3/2` najde jenom první výskyt, proměnné neváže (hledá pouze prvky, které jsou identické jako termy).

Řešení 12.1.3 V níže uvedených výpočetních stromech byly pro zpřehlednění vynechány predikáty, které pouze unifikují proměnnou s číselnou hodnotou (a uspějí s triviální substitucí čísla za onu proměnnou).

a)



b)



Řešení 12.1.4

```
insert(X, [], [X]).
insert(X, [H|T], R) :- X =< H, !, R = [X,H|T].
insert(X, [H|T], [H|T1]) :- insert(X, T, T1).
```

Všimněte si, že pokud bychom namísto `R` v druhé klauzuli umístili výraz `[X,H|T]`, máme stejný problém jako u `max/3` v příkladu 12.1.1: například dotaz `?- insert(1, [1,2], [1,2,1]).` uspěje.

Řešení 12.1.5

```
remove(_X, [], []).
remove(X, [X|S], S1) :- !, remove(X, S, S1).
remove(X, [Y|S], [Y|S1]) :- remove(X, S, S1).
```

Zvažte, proč je nutný řez a jak by vypadal výsledek bez něj.

Řešení 12.1.6 V řešení využíváme knihovní funkci `member/2`, která uspěje, když je první argument prvkem seznamu v druhém argumentu.

```
intersection([], _List, []).
intersection([H|T], List, R) :-
    member(H, List), !, R = [H|Rest], intersection(T, List, Rest).
intersection([_H|T], List, Rest) :- intersection(T, List, Rest).

difference([], _List, []).
difference([H|T], List, Rest) :-
    member(H, List), !, difference(T, List, Rest).
difference([H|T], List, [H|Rest]) :- difference(T, List, Rest).
```

Řešení 12.1.7

```
fib(N, 1) :- N =< 2, !.
fib(N, X) :- N1 is N - 2, fib(N1, 1, 1, X).
fib(0, _A, X, X) :- !.
fib(N, A, B, X) :- N1 is N - 1, C is A + B, fib(N1, B, C, X).
```

Řešení 12.1.8 Řez doplníme na konec sedmého řádku. Ten bude tedy vypadat následovně:

```
s(X) :- t(X, a), !.
```

Řešení 12.2.1

```
nd35a(X) :- X mod 5 =\= 0, X mod 3 =\= 0.

nd35b(X) :- X mod 5 =:= 0, !, fail.
nd35b(X) :- X mod 3 =:= 0, !, fail.
nd35b(X).

nd35c(X) :- \+(X mod 3 =:= 0), \+(X mod 5 =:= 0).
```

Řešení 12.3.1 Uvažme predikáty `variation3/2` a `combination3/2` z řešení úlohy 11.2.6. Pak už je definice poměrně jednoduchá:

a)
```
variation3all(List, Y) :-
    findall(Variation3, variation3(List, Variation3), Y).
```

b)
```
combination3all(List, Y) :-
    findall(Combination3, combination3(List, Combination3), Y).
```

Řešení 12.3.2

a) `List = [b,c,d].`
b) `List = [a,g].`
c) `List = [a,a,a,e,g,g,i].`
d)
```
Y = a, List = [i];
Y = b, List = [a,g];
Y = c, List = [a,e];
Y = d, List = [a];
Y = h, List = [g].
```
e) `List = [a,e,g,i].`

Řešení 12.3.3

```
subsets(X, S) :- isset(X), findall(Y, subset(X, Y), S).

isset([]).
isset([X|T]) :- \+ member(X, T), isset(T).

subset([], []).
subset([_X|T], S) :- subset(T, S).
subset([X|T], [X|S]) :- subset(T, S).
```

Řešení 12.4.1

```
fileSum(FileName, Sum) :-
    seeing(Old), seen, see(FileName),
    read(X),
    fileSumH(X, 0, Sum),
    !,
    seen, see(Old).

fileSumH(end_of_file, Sum, Sum).
fileSumH(s(N), AcSum, Sum) :-
    AcSum1 is AcSum + N,
    read(X),
    fileSumH(X, AcSum1, Sum).
fileSumH(_, AcSum, Sum) :-
    read(X),
    fileSumH(X, AcSum, Sum).
```

Řez v řešení je nutný, aby nedošlo k backtrackování zpět do `fileSumH/3` v případě, že by nějaký pozdější cíl selhal (což by způsobilo čtení z jiného proudu!).

Řešení 12.4.2 Nejdříve si zapamatujeme, který proud byl původně nastaven jako čtecí (abychom ho pak mohli znova nastavit, až skončíme). Pak proud zavřeme a otevřeme požadovaný soubor.

Pak opakujeme (predikát `repeat/0`) načtení znaku, test na konec souboru a unifikaci na požadovaný znak. Jestli test na konec souboru uspěl, výpočet zařízneme, obnovíme původní vstupní proud a selžeme (znak se nenašel).

```
contains(FileName, Char) :- seeing(Old), seen, see(FileName),
    repeat, get_char(X),
    (X == end_of_file -> !, restore(Old), fail ; Char = X).

restore(Old) :- seen, see(Old).
```

Řešení 12.4.3 Stačilo by nám využít následujícího dotazu:

```
findall(Char, contains(FileName, Char), List).
```

Uvedené řešení však načítá znaky ze souboru postupně, po jednom. Navíc pro získání každého znaku vyhodnocuje znovu predikát `contains/2`. Je lepší využít efektivní knihovní funkci `read_file_to_codes/3`.

Řešení 13.1.1 Řešení definuje predikát `puzzle/1`, který má na vstupu term představující náš algebrogram. Výpočet spustíte třeba dotazem `?- puzzle(X).`

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    M #\= 0, S #\= 0,
    all_different(Vars),
    S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    label(Vars).
```

Řešení 13.1.2 Řešení obsahuje pomocný predikát `printAll/1`, který vypíše všechny prvky zadaného seznamu a pak zalomí řádek. Samotný součet využívá proměnné `Donald`, `Gerald` a `Robert`, které jsou definovány pomocí predikátu `scalar_product/4`.

```
puzzle2([D,O,N,A,L,G,E,R,B,T]) :-
    [O,N,A,L,E,B,T] ins 0..9,
    [D,G,R] ins 1..9,
    [Donald, Gerald, Robert] ins 0..1000000,
    all_distinct([D,O,N,A,L,G,E,R,B,T]),
    scalar_product([100000,10000,1000,100,10,1], [D,O,N,A,L,D],
                   #=, Donald),
    scalar_product([100000,10000,1000,100,10,1], [G,E,R,A,L,D],
                   #=, Gerald),
    scalar_product([100000,10000,1000,100,10,1], [R,O,B,E,R,T],
                   #=, Robert),
    Donald + Gerald #= Robert,
    label([D,O,N,A,L,G,E,R,B,T]),
    printAll([D,O,N,A,L,D]),
    print(+), nl,
    printAll([G,E,R,A,L,D]),
```

```
    print(=), nl,
    printAll([R,O,B,E,R,T]).

printAll([]) :- nl.
printAll([H|T]) :- print(H), printAll(T).
```

Řešení 13.1.3

```
puzzle3([K,C,I,O,A,M,L]) :-
    [K,I,O,A,L] ins 1..9,
    [C,M] ins 0..9,
    all_distinct([K,C,I,O,A,M,L]),
    10*K + C + I #= 10*O + K,
    A + A #= 10*K + M,
    10*O + L + 10*K + O #= 10*L + I,
    10*K + C + A #= 10*O + L,
    I + A #= 10*K + O,
    10*O + K + 10*K + M #= 10*L + I,
    label([K,C,I,O,A,M,L]).
```

Řešení 13.1.4

```
fact(0, 1).
fact(N, F) :- N #> 0, N1 #= N - 1, F #= N * F1, fact(N1, F1).
```

Výhodou této formulace je, že je intuitivní a zároveň silnější než běžná definice, protože můžeme pokládat dokonce i dotazy jako `?- fact(N, 120).` nebo dokonce `?- fact(X, Y).` Také pořadí podcílů je volnější.

Řešení 13.1.5 Seznam `Value` představuje hodnoty mincí, které máme k dispozici. Seznam `Count` představuje množství mincí jednotlivých hodnot, které potřebujeme na získání sumy `Sum`. Řešení můžeme odzkoušet třeba na dotazu `?- coins([1,2,5,10,20,50], 174, X).`

```
coins(Value, Sum, Count) :-
    length(Value, Len),
    length(Count, Len),
    Count ins 0..Sum,
    scalar_product(Value, Count, #=, Sum),
    label(Count).
```

Řešení lze vylepšit tak, abychom nejdříve získali řešení s nejmenším počtem mincí. Na to si zavedeme pomocnou proměnnou `Coins` s celkovým počtem mincí a pomocí predikátu `labeling/2` řekneme, že jí chceme minimalizovat:

```
coins(Value, Sum, Count) :-
    length(Value, Len),
    length(Count, Len),
    Count ins 0..Sum,
    scalar_product(Value, Count, #=, Sum),
    sum(Count, #=, Coins),
    labeling([min(Coins)], Count).
```

Řešení 13.1.6

```
:- use_module(library(clpfd)).

queens(N, L, Type) :-
   length(L, N),
   L ins 1..N,
   constr_all(L),
   labeling(Type, L).

constr_all([]).
constr_all([X|Xs]) :- constr_between(X, Xs, 1), constr_all(Xs).

constr_between(_, [], _).
constr_between(X, [Y|Ys], N) :-
   no_threat(X, Y, N),
   N1 is N + 1,
   constr_between(X, Ys, N1).

no_threat(X, Y, I):- X #\= Y, X + I #\= Y, X - I #\= Y.
```

Řešení 13.1.7

```
sudoku4([A1,A2,A3,A4,
         B1,B2,B3,B4,
         C1,C2,C3,C4,
         D1,D2,D3,D4]) :-
    Values = [A1,A2,A3,A4,B1,B2,B3,B4,C1,C2,C3,C4,D1,D2,D3,D4],
    Values ins 1..4,
    all_distinct([A1,A2,A3,A4]),
    all_distinct([B1,B2,B3,B4]),
    all_distinct([C1,C2,C3,C4]),
    all_distinct([D1,D2,D3,D4]),
    all_distinct([A1,B1,C1,D1]),
    all_distinct([A2,B2,C2,D2]),
    all_distinct([A3,B3,C3,D3]),
    all_distinct([A4,B4,C4,D4]),
    all_distinct([A1,A2,B1,B2]),
    all_distinct([A3,A4,B3,B4]),
    all_distinct([C1,C2,D1,D2]),
    all_distinct([C3,C4,D3,D4]),
    label(Values),
    printSudoku(Values).

printSudoku([]) :- print(+-+-+-+-+), nl.
printSudoku([V1,V2,V3,V4|Rest]) :-
    print('+-+-+-+-+'), nl,
    print('|'), print(V1), print('|'), print(V2), print('|'),
    print(V3), print('|'), print(V4), print('|'), nl,
```

```
    printSudoku(Rest).
```

Řešení 13.2.1

a) Ano, dojde k unifikaci se substitucí `X = 1`.
b) Ne, `X` je neinstanciována proměnná, `1` je číslo/atom, tedy nejde o stejné termy. Unifikace se při `==` neprovádí.
c) Dojde k chybě -- při aritmetickém porovnání musí být obě strany plně instanciovány.
d) Ano, dojde k aritmetickému vyhodnocení pravé strany a unifikaci výsledku `2` s proměnnou `X`.
e) Ano, po unifikaci `X` s `1` porovnání termů uspěje, protože porovnání bere ohled na dříve provedené substituce.
f) Ne, nedojde k aritmetickému vyhodnocení, a tedy tyto výrazy představují různé termy.
g) Ano, dojde k aritmetickému vyhodnocení a porovnání uspěje.
h) Ano, se substitucí `Z = z(X), X = Y`.
i) Ne, nelze použít proměnnou na místě funktoru. (I když tento cíl lze dosáhnout pomocí predikátů `functor/3`, `arg/3` nebo `=../2`).
j) Ne, jde o různé funktory.

Řešení 13.2.2 Prvním problémem je, že program nelze přeložit a způsobuje ho nesprávné použití operátoru `=`, který jenom unifikuje obě strany, ale aritmeticky nevyhodnocuje. Nahradíme ho tedy `is`.

Dále, takto napsaný program nikdy nekončí, protože vždy se použije pravidlo a k ukončujícímu faktu nikdy nedojde. Je tedy třeba prohodit fakt a pravidlo.

Teď už dostaneme výsledek. Avšak program stále nefunguje například pro dotaz `?- fact(1, 2).` V tomto případě se nikdy nepoužije fakt. Musíme tedy doplnit omezující podmínku, že první argument musí být v pravidle větší než `0`.

Zůstává poslední neduh. Pokud zadáme dotaz, Prolog se pokouší po vrácení prvního řešení nalézt další, avšak víme, že žádné další nebude existovat. Proto použijeme řez a upravíme ním fakt pro faktoriál nuly.

Výslední program bude následovný:

```
fact(0, 1) :- !.
fact(N, Fact) :- N > 0, M is N - 1, fact(M, FactP), Fact is N * FactP.
```

Řešení 13.2.3

```
merge([], X, X) :- !.
merge(X, [], X) :- !.
merge([H1|T1], [H2|T2], [H1|T]) :- H1 =< H2, !, merge(T1, [H2|T2], T).
merge([H1|T1], [H2|T2], [H2|T]) :- merge([H1|T1], T2, T).

split([], [], []).
split([X], [X], []).
split([X,Y|Rest], [X|Rest1], [Y|Rest2]) :- split(Rest, Rest1, Rest2).

mergesort([], []) :- !.
mergesort([X], [X]) :- !.
mergesort(List, Sorted) :-
```

```
    split(List, SubList1, SubList2),
    mergesort(SubList1, Sub1Sorted),
    mergesort(SubList2, Sub2Sorted),
    merge(Sub1Sorted, Sub2Sorted, Sorted).
```

Řešení 13.2.4

```
split(_Pivot, [], [], []) :- !.
split(Pivot, [Head|List], [Head|Small], Big) :-
    Head < Pivot,
    !,
    split(Pivot, List, Small, Big).
split(Pivot, [Head|List], Small, [Head|Big]) :-
    split(Pivot, List, Small, Big).

quicksort([], []).
quicksort([X|XS], Sorted) :-
    split(X, XS, Small, Big),
    quicksort(Small, SmallSorted),
    quicksort(Big, BigSorted),
    append(SmallSorted, [X|BigSorted], Sorted).
```

Řešení 13.2.5 Řešení zadaných úloh najdete v souboru *einsteinSol.pl*. Řešení využívající jiné kódování najdete v souboru *einsteinSol2.pl*. Soubory naleznete v ISu a v příloze sbírky.

Řešení 13.2.6

```
equals([], []) :- !.
equals([X|T1], [X|T2]) :- !, equals(T1, T2).
equals([], _S) :- write('1st list is shorter'), !, fail.
equals(_S, []) :- write('2nd list is shorter'), !, fail.
equals([X|_T1], [Y|_T2]) :- write(X), write(' does not equal '),
    write(Y), !, fail.
```

Pro úplnost dodejme, že jestli není vyžadován důvod nerovnosti, vystačili bychom si s pouhým faktem `equals(S, S).` (avšak pouze v případě, že `S` neobsahuje proměnné).

Řešení 13.2.7 Úloha bez rozšíření je v podstatě ekvivalentní úloze o rodinných vztazích z dřívějších cvičení.

```
trip(A, A) :- !.
trip(A, B) :- road(A, X), trip(X, B).
```

Následující řešení zahrnuje první rozšíření:

```
trip(A, A, [A]).
trip(A, B, [A|S]) :- road(A, X), trip(X, B, S).
```

Prezentované řešení pro druhé rozšíření využívá tzv. dynamické klauzule. Ty dovolují měnit programovou databázi za běhu přidáváním (`assert/1`) nebo odebíráním (`retract/1`) nových klauzulí. Alternativním řešením by bylo přidat další argument -- seznam navštívených míst.

```
:- dynamic visited/1.
trip(A, A, [A]).
trip(A, B, [A|S]) :- assert(visited(A)), road(A, X),
    \+ visited(X), trip(X, B, S).
trip(A, _B, _S) :- retract(visited(A)), fail.
```

A Přílohy

A.1 pt.hs

```
main :: IO ()
main =
  do putStr "pocet radku: "                                  -- prompt
     n <- getLine >>= return . read                          -- nacteni
   cisla
     if n > 0
        then do putStr (unlines (map showLine (take n pt)))  -- vypis
   trojuhelnika
                main                                         --   a
   opakovani
        else return ()                                       -- ukonceni

pt :: [[Integer]]                                -- Pascaluv trojuhelnik
pt  = iterate (\r -> zipWith (+) ([0]++r) (r++[0])) [1]

showLine   :: [Integer] -> String                        -- vypis radku
showLine r = replicate m ' ' ++ concat (map showBox r)
             where m = abs (ll - length r * bwi) `div` 2

showBox  :: Integer -> String                            -- vypis prvku
showBox k = replicate p ' ' ++ sk ++ replicate r ' '
            where  sk = show k
                   l  = bwi - length sk
                   r = l `div` 2
                   p = l - r

bwi, ll :: Int
bwi = 5        -- max sirka cisla
ll = 80        -- delka radku na vystupu
```

A.2 guess.hs

```
import Char
```

```
main = guess 1 10

query ot = do putStr ot
              ans <- getLine
              if (ans == "ano") then return True else return False

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
```

A.3 BinTree.hs

```
module BinTree (BinTree(..), drawTree) where

-- IB015 binarni stromy
-- funkce na vykresleni stromu BinTree a pomoci externi knihovny Data.Tree

-- externi knihovna Data.Tree pracuje s n-arnymi stromy
-- vice info na
   http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Tree.ht
import qualified Data.Tree as T (Tree(..), drawTree)

-- nase definice binarniho stromu
data BinTree a = Empty | Node a (BinTree a) (BinTree a) deriving (Eq, Show)

-- transformuje nase stromy na typ, se ktorym pracuje knihovna
convertTree :: Show a => BinTree a -> T.Tree String
convertTree Empty = T.Node "Empty" []
convertTree (Node x l r) = T.Node (show x) (map convertTree [l,r])

-- samotna vykreslovaci funkce
drawTree :: Show a => BinTree a -> IO ()
drawTree = putStr . T.drawTree . convertTree
```

A.4 treeFold.hs

```
data BinTree a = Node a (BinTree a) (BinTree a)
```

```
              | Empty
              deriving (Eq, Show)

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

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

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

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

tree04 :: BinTree a
tree04 = Empty

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

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

A.5 pedigree.pl

```
woman(eva).
woman(lenka).
woman(pavla).
woman(jana).
woman(vera).

man(petr).
man(filip).
man(pavel).
man(jan).
man(adam).
man(tomas).
```

```
man(michal).
man(radek).

parent(petr, filip).
parent(petr, lenka).
parent(pavel, jan).
parent(adam, petr).
parent(tomas, michal).
parent(michal, radek).
parent(pavel, tomas).
parent(eva, filip).
parent(jana, lenka).
parent(pavla, petr).
parent(pavla, tomas).
parent(lenka, vera).

father(Father, Child) :- parent(Father, Child), man(Father).
```

A.6 einstein.pl

```
barva(bila).
barva(cervena).
barva(modra).
barva(zelena).
barva(zluta).

narod(brit).
narod(dan).
narod(nemec).
narod(nor).
narod(sved).

zver(pes).
zver(ptak).
zver(kocka).
zver(kun).
zver(rybicky).

piti(caj).
piti(kava).
piti(mleko).
piti(pivo).
piti(voda).

kouri(pallmall).
kouri(dunhill).
```

```
kouri(blend).
kouri(bluemaster).
kouri(prince).

rule1(B1,B2,B3,B4,B5,N1,N2,N3,N4,N5)
 :- B1 = cervena, N1 = brit;
    B2 = cervena, N2 = brit;
    B3 = cervena, N3 = brit;
    B4 = cervena, N4 = brit;
    B5 = cervena, N5 = brit.

rule2(N1,N2,N3,N4,N5,Z1,Z2,Z3,Z4,Z5)
 :- Z1 = pes, N1 = sved;
    Z2 = pes, N2 = sved;
    Z3 = pes, N3 = sved;
    Z4 = pes, N4 = sved;
    Z5 = pes, N5 = sved.

rule3(N1,N2,N3,N4,N5,P1,P2,P3,P4,P5)
 :- N1 = dan, P1 = caj;
    N2 = dan, P2 = caj;
    N3 = dan, P3 = caj;
    N4 = dan, P4 = caj;
    N5 = dan, P5 = caj.

rule4(B1,B2,B3,B4,B5)
 :- B1 = zelena, B2 = bila;
    B2 = zelena, B3 = bila;
    B3 = zelena, B4 = bila;
    B4 = zelena, B5 = bila.

rule5(B1,B2,B3,B4,B5,P1,P2,P3,P4,P5)
 :- B1 = zelena, P1 = kava;
    B2 = zelena, P2 = kava;
    B3 = zelena, P3 = kava;
    B4 = zelena, P4 = kava;
    B5 = zelena, P5 = kava.

rule6(K1,K2,K3,K4,K5,Z1,Z2,Z3,Z4,Z5)
 :- K1 = pallmall, Z1 = ptak;
    K2 = pallmall, Z2 = ptak;
    K3 = pallmall, Z3 = ptak;
    K4 = pallmall, Z4 = ptak;
    K5 = pallmall, Z5 = ptak.

rule7(B1,B2,B3,B4,B5,K1,K2,K3,K4,K5)
 :- B1 = zluta, K1 = dunhill;
```

```
    B2 = zluta, K2 = dunhill;
    B3 = zluta, K3 = dunhill;
    B4 = zluta, K4 = dunhill;
    B5 = zluta, K5 = dunhill.

rule10(K1,K2,K3,K4,K5,Z1,Z2,Z3,Z4,Z5)
 :- K1 = blend, Z2 = kocka;
    K2 = blend, Z1 = kocka;
    K2 = blend, Z3 = kocka;
    K3 = blend, Z2 = kocka;
    K3 = blend, Z4 = kocka;
    K4 = blend, Z3 = kocka;
    K4 = blend, Z5 = kocka;
    K5 = blend, Z4 = kocka.

rule11(K1,K2,K3,K4,K5,Z1,Z2,Z3,Z4,Z5)
 :- K1 = dunhill, Z2 = kun;
    K2 = dunhill, Z1 = kun;
    K2 = dunhill, Z3 = kun;
    K3 = dunhill, Z2 = kun;
    K3 = dunhill, Z4 = kun;
    K4 = dunhill, Z3 = kun;
    K4 = dunhill, Z5 = kun;
    K5 = dunhill, Z4 = kun.

rule12(K1,K2,K3,K4,K5,P1,P2,P3,P4,P5)
 :- K1 = bluemaster, P1 = pivo;
    K2 = bluemaster, P2 = pivo;
    K3 = bluemaster, P3 = pivo;
    K4 = bluemaster, P4 = pivo;
    K5 = bluemaster, P5 = pivo.


rule13(K1,K2,K3,K4,K5,N1,N2,N3,N4,N5)
 :- K1 = prince, N1 = nemec;
    K2 = prince, N2 = nemec;
    K3 = prince, N3 = nemec;
    K4 = prince, N4 = nemec;
    K5 = prince, N5 = nemec.

rule14(B1,B2,B3,B4,B5,N1,N2,N3,N4,N5)
 :- B1 = modra, N2 = nor;
    B2 = modra, N1 = nor;
    B2 = modra, N3 = nor;
    B3 = modra, N2 = nor;
    B3 = modra, N4 = nor;
    B4 = modra, N3 = nor;
```

```
    B4 = modra, N5 = nor;
    B5 = modra, N4 = nor.


rule15(K1,K2,K3,K4,K5,P1,P2,P3,P4,P5)
 :- K1 = blend, P2 = voda;
    K2 = blend, P1 = voda;
    K2 = blend, P3 = voda;
    K3 = blend, P2 = voda;
    K3 = blend, P4 = voda;
    K4 = blend, P3 = voda;
    K4 = blend, P5 = voda;
    K5 = blend, P4 = voda.

reseni(B1,B2,B3,B4,B5,N1,N2,N3,N4,N5,Z1,Z2,Z3,Z4,Z5,P1,P2,P3,P4,P5,K1,K2,K3,K4,K5)
   :-
    barva(B1),
    barva(B2), B2\=B1,
    barva(B3), B3\=B1, B3\=B2,
    barva(B4), B4\=B1, B4\=B2, B4\=B3,
    barva(B5), B5\=B1, B5\=B2, B5\=B3, B5\=B4,
    narod(N1),
    narod(N2), N2\=N1,
    narod(N3), N3\=N1, N3\=N2,
    narod(N4), N4\=N1, N4\=N2, N4\=N3,
    narod(N5), N5\=N1, N5\=N2, N5\=N3, N5\=N4,
    zver(Z1),
    zver(Z2), Z2\=Z1,
    zver(Z3), Z3\=Z1, Z3\=Z2,
    zver(Z4), Z4\=Z1, Z4\=Z2, Z4\=Z3,
    zver(Z5), Z5\=Z1, Z5\=Z2, Z5\=Z3, Z5\=Z4,
    kouri(K1),
    kouri(K2), K2\=K1,
    kouri(K3), K3\=K1, K3\=K2,
    kouri(K4), K4\=K1, K4\=K2, K4\=K3,
    kouri(K5), K5\=K1, K5\=K2, K5\=K3, K5\=K4,
    piti(P1),
    piti(P2), P2\=P1,
    piti(P3), P3\=P1, P3\=P2,
    piti(P4), P4\=P1, P4\=P2, P4\=P3,
    piti(P5), P5\=P1, P5\=P2, P5\=P3, P5\=P4,
/*  1   Brit bydli v cervenem dome. */
    rule1(B1,B2,B3,B4,B5,N1,N2,N3,N4,N5),
/*  2   Sved chova psa. */
    rule2(N1,N2,N3,N4,N5,Z1,Z2,Z3,Z4,Z5),
/*  3   Dan pije caj. */
    rule3(N1,N2,N3,N4,N5,P1,P2,P3,P4,P5),
```

```
/*  4   Zeleny dum stoji hned nalevo od bileho. */
    rule4(B1,B2,B3,B4,B5),
/*  5   Majitel zeleneho domu pije kavu. */
    rule5(B1,B2,B3,B4,B5,P1,P2,P3,P4,P5),
/*  6   Ten, kdo kouri PallMall, chova ptaka. */
    rule6(K1,K2,K3,K4,K5,Z1,Z2,Z3,Z4,Z5),
/*  7   Majitel zluteho domu kouri Dunhill. */
    rule7(B1,B2,B3,B4,B5,K1,K2,K3,K4,K5),
/*  8   Ten, kdo bydli uprostred rady domu, pije mleko. */
    P3 = mleko,
/*  9   Nor bydli v prvnim dome. */
    N1 = nor,
/*  10  Ten, kdo kouri Blend, bydli vedle toho, kdo chova kocku. */
    rule10(K1,K2,K3,K4,K5,Z1,Z2,Z3,Z4,Z5),
/*  11  Ten, kdo chova kone, bydli vedle toho, kdo kouri Dunhill. */
    rule11(K1,K2,K3,K4,K5,Z1,Z2,Z3,Z4,Z5),
/*  12  Ten, kdo kouri BlueMaster, pije pivo. */
    rule12(K1,K2,K3,K4,K5,P1,P2,P3,P4,P5),
/*  13  Nemec kouri Prince. */
    rule13(K1,K2,K3,K4,K5,N1,N2,N3,N4,N5),
/*  14  Nor bydli vedle modreho domu. */
    rule14(B1,B2,B3,B4,B5,N1,N2,N3,N4,N5),
/*  15  Ten, kdo kouri Blend, ma souseda, ktery pije vodu. */
    rule15(K1,K2,K3,K4,K5,P1,P2,P3,P4,P5).

rybicky(X) :-
  reseni(_,_,_,_,_,X,_,_,_,_,rybicky,_,_,_,_,_,_,_,_,_,_,_,_,_,_);
  reseni(_,_,_,_,_,_,X,_,_,_,_,rybicky,_,_,_,_,_,_,_,_,_,_,_,_,_);
  reseni(_,_,_,_,_,_,_,X,_,_,_,_,rybicky,_,_,_,_,_,_,_,_,_,_,_,_);
  reseni(_,_,_,_,_,_,_,_,X,_,_,_,_,rybicky,_,_,_,_,_,_,_,_,_,_,_);
  reseni(_,_,_,_,_,_,_,_,_,X,_,_,_,_,rybicky,_,_,_,_,_,_,_,_,_,_).
```

A.7 einsteinSol.pl

```
isLeftTo(X,Y,SX,SY)
 :- SX = [X,_,_,_,_], SY = [_,Y,_,_,_];
    SX = [_,X,_,_,_], SY = [_,_,Y,_,_];
    SX = [_,_,X,_,_], SY = [_,_,_,Y,_];
    SX = [_,_,_,X,_], SY = [_,_,_,_,Y].

isNextTo(X,Y,SX,SY)
 :- isLeftTo(X,Y,SX,SY);
    isLeftTo(Y,X,SY,SX).

isTogetherWith(X,Y,SX,SY)
 :- SX = [X,_,_,_,_], SY = [Y,_,_,_,_];
```

```
    SX = [_,X,_,_,_], SY = [_,Y,_,_,_];
    SX = [_,_,X,_,_], SY = [_,_,Y,_,_];
    SX = [_,_,_,X,_], SY = [_,_,_,Y,_];
    SX = [_,_,_,_,X], SY = [_,_,_,_,Y].

reseni(B,N,Z,P,K) :-
/*  8   Ten, kdo bydli uprostred rady domu, pije mleko. */
    P = [_,_,mleko,_,_],
/*  9   Nor bydli v prvnim dome. */
    N = [nor,_,_,_,_],
/*  4   Zeleny dum stoji hned nalevo od bileho. */
    isLeftTo(zelena,bila,B,B),
/*  1   Brit bydli v cervenem dome. */
    isTogetherWith(brit,cervena,N,B),
/*  14  Nor bydli vedle modreho domu. */
    isNextTo(nor,modra,N,B),
/*  2   Sved chova psa. */
    isTogetherWith(sved,pes,N,Z),
/*  6   Ten, kdo kouri PallMall, chova ptaka. */
    isTogetherWith(pallmall,ptak,K,Z),
/*  7   Majitel zluteho domu kouri Dunhill. */
    isTogetherWith(zluta,dunhill,B,K),
/*  10  Ten, kdo kouri Blend, bydli vedle toho, kdo chova kocku. */
    isNextTo(blend,kocka,K,Z),
/*  11  Ten, kdo chova kone, bydli vedle toho, kdo kouri Dunhill. */
    isNextTo(kun,dunhill,Z,K),
/*  13  Nemec kouri Prince. */
    isTogetherWith(nemec,prince,N,K),
/*  5   Majitel zeleneho domu pije kavu. */
    isTogetherWith(zelena,kava,B,P),
/*  3   Dan pije caj. */
    isTogetherWith(dan,caj,N,P),
/*  12  Ten, kdo kouri BlueMaster, pije pivo. */
    isTogetherWith(bluemaster,pivo,K,P),
/*  15  Ten, kdo kouri Blend, ma souseda, ktery pije vodu. */
    isNextTo(blend,voda,K,P).

rybicky(X) :- reseni(_,N,Z,_,_), isTogetherWith(X,rybicky,N,Z).
```

A.8 einsteinSol2.pl

```
/* Reprezentujme jeden dum jako seznam [Barva,Narodnost,Kurivo,Piti,Zvire]
 a vsechny domy jako seznam peti takovych seznamu (v poradi zleva doprava,
 jak to vidi pozorovatel). */

/* Budeme vyuzivat obousmernost predikatu; member umoznuje nejen
```

```
 testovat, zda je prvek v seznamu, ale taky ho umi "vsunout" do seznamu
 na misto promenne. */
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

% potrebujeme urcit, co je nalevo (z pohledu pozorovatele)
left_right(L,R,[L,R,_,_,_]).
left_right(L,R,[_,L,R,_,_]).
left_right(L,R,[_,_,L,R,_]).
left_right(L,R,[_,_,_,L,R]).

% potrebujeme urcit, co je vedle
next_to(X,Y,L) :- left_right(X,Y,L).
next_to(X,Y,L) :- left_right(Y,X,L).

% Lustime zebru, kodujeme podminky ze zadani, nejprve ty nejvice urcujici.
% Na poradi nekterych podminek zalezi efektivita.
% To, ze barvy, narodnosti, kurivo, napoje a zvirata jsou exkluzivni,
% zajistuji jejich pocty v zadani, zvolena reprezentace a formulace
   podminek.
% Na miste pateho zvirete zustane volna promenna.
zebra(S) :-
   % [Barva,Narodnost,Kurivo,Piti,Zvire] pro pripomenuti
   S = [[_,nor,_,_,_],_,[_,_,_,mleko,_],_,_],          % 8 a 9
   next_to([_,nor,_,_,_],[modry,_,_,_,_],S),           % 14
   member([cerveny,brit,_,_,_],S),                     % 1
   member([_,sved,_,_,pes],S),                         % 2
   member([_,dan,_,caj,_],S),                          % 3
   left_right([zeleny,_,_,_,_],[bily,_,_,_,_],S),      % 4
   member([zeleny,_,_,kava,_],S),                      % 5
   member([_,_,pallmall,_,ptak],S),                    % 6
   member([zluty,_,dunhill,_,_],S),                    % 7
   next_to([_,_,blend,_,_],[_,_,_,_,kocka],S),         % 10
   next_to([_,_,_,_,kun],[_,_,dunhill,_,_],S),         % 11
   member([_,_,bluemaster,pivo,_],S),                  % 12
   member([_,nemec,prince,_,_],S),                     % 13
   next_to([_,_,blend,_,_],[_,_,_,voda,_],S).          % 15

% kdo ma rybicky?
rybicky(X) :- zebra(S), member([_,X,_,_,rybicky],S).

% pomoci ?- zebra([A,B,C,D,E]). celkem citelne vypiseme, jak cela situace
% vypada. Kdyby existovalo vice reseni, pomoci stredniku najdeme vsechna.
```

A.9 sudoku.pl

```
% sudoku.pl
% Aswin F. van Woudenberg
% to run type: swipl -f sud.pl -g solve_sudoku -t halt -q


:- use_module(library('clp/bounds')).

solve_sudoku :-
read_sudoku(L),
sudoku(L),
write_sol(L).

sudoku(Vars) :-
Vars = [
AA, AB, AC, AD, AE, AF, AG, AH, AI,
BA, BB, BC, BD, BE, BF, BG, BH, BI,
CA, CB, CC, CD, CE, CF, CG, CH, CI,
DA, DB, DC, DD, DE, DF, DG, DH, DI,
EA, EB, EC, ED, EE, EF, EG, EH, EI,
FA, FB, FC, FD, FE, FF, FG, FH, FI,
GA, GB, GC, GD, GE, GF, GG, GH, GI,
HA, HB, HC, HD, HE, HF, HG, HH, HI,
IA, IB, IC, ID, IE, IF, IG, IH, II],
Vars in 1..9,
all_different([AA, AB, AC, BA, BB, BC, CA, CB, CC]),
all_different([AD, AE, AF, BD, BE, BF, CD, CE, CF]),
all_different([AG, AH, AI, BG, BH, BI, CG, CH, CI]),
all_different([DA, DB, DC, EA, EB, EC, FA, FB, FC]),
all_different([DD, DE, DF, ED, EE, EF, FD, FE, FF]),
all_different([DG, DH, DI, EG, EH, EI, FG, FH, FI]),
all_different([GA, GB, GC, HA, HB, HC, IA, IB, IC]),
all_different([GD, GE, GF, HD, HE, HF, ID, IE, IF]),
all_different([GG, GH, GI, HG, HH, HI, IG, IH, II]),
all_different([AA, AB, AC, AD, AE, AF, AG, AH, AI]),
all_different([BA, BB, BC, BD, BE, BF, BG, BH, BI]),
all_different([CA, CB, CC, CD, CE, CF, CG, CH, CI]),
all_different([DA, DB, DC, DD, DE, DF, DG, DH, DI]),
all_different([EA, EB, EC, ED, EE, EF, EG, EH, EI]),
all_different([FA, FB, FC, FD, FE, FF, FG, FH, FI]),
all_different([GA, GB, GC, GD, GE, GF, GG, GH, GI]),
all_different([HA, HB, HC, HD, HE, HF, HG, HH, HI]),
all_different([IA, IB, IC, ID, IE, IF, IG, IH, II]),
all_different([AA, BA, CA, DA, EA, FA, GA, HA, IA]),
all_different([AB, BB, CB, DB, EB, FB, GB, HB, IB]),
all_different([AC, BC, CC, DC, EC, FC, GC, HC, IC]),
all_different([AD, BD, CD, DD, ED, FD, GD, HD, ID]),
all_different([AE, BE, CE, DE, EE, FE, GE, HE, IE]),
```

```
all_different([AF, BF, CF, DF, EF, FF, GF, HF, IF]),
all_different([AG, BG, CG, DG, EG, FG, GG, HG, IG]),
all_different([AH, BH, CH, DH, EH, FH, GH, HH, IH]),
all_different([AI, BI, CI, DI, EI, FI, GI, HI, II]),
label(Vars).

write_sol([
AA, AB, AC, AD, AE, AF, AG, AH, AI,
BA, BB, BC, BD, BE, BF, BG, BH, BI,
CA, CB, CC, CD, CE, CF, CG, CH, CI,
DA, DB, DC, DD, DE, DF, DG, DH, DI,
EA, EB, EC, ED, EE, EF, EG, EH, EI,
FA, FB, FC, FD, FE, FF, FG, FH, FI,
GA, GB, GC, GD, GE, GF, GG, GH, GI,
HA, HB, HC, HD, HE, HF, HG, HH, HI,
IA, IB, IC, ID, IE, IF, IG, IH, II]) :-

write('|'), write(AA), write(AB), write(AC), write('|'),write(AD),
   write(AE), write(AF), write('|'),write(AG), write(AH), write(AI),
   write('|'),nl,
write('|'), write(BA), write(BB), write(BC), write('|'),write(BD),
   write(BE), write(BF), write('|'),write(BG), write(BH), write(BI),
   write('|'),nl,
write('|'), write(CA), write(CB), write(CC), write('|'),write(CD),
   write(CE), write(CF), write('|'),write(CG), write(CH), write(CI),
   write('|'),nl,
write('+---+---+---+'), nl,
write('|'), write(DA), write(DB), write(DC), write('|'),write(DD),
   write(DE), write(DF), write('|'),write(DG), write(DH), write(DI),
   write('|'),nl,
write('|'), write(EA), write(EB), write(EC), write('|'),write(ED),
   write(EE), write(EF), write('|'),write(EG), write(EH), write(EI),
   write('|'),nl,
write('|'), write(FA), write(FB), write(FC), write('|'),write(FD),
   write(FE), write(FF), write('|'),write(FG), write(FH), write(FI),
   write('|'),nl,
write('+---+---+---+'), nl,
write('|'), write(GA), write(GB), write(GC), write('|'),write(GD),
   write(GE), write(GF), write('|'),write(GG), write(GH), write(GI),
   write('|'),nl,
write('|'), write(HA), write(HB), write(HC), write('|'),write(HD),
   write(HE), write(HF), write('|'),write(HG), write(HH), write(HI),
   write('|'),nl,
write('|'), write(IA), write(IB), write(IC), write('|'),write(ID),
   write(IE), write(IF), write('|'),write(IG), write(IH), write(II),
   write('|'),nl,
write('+---+---+---+'), nl.
```

```
get_num(NumValue) :-
get_single_char(Value),
(member(Value, [49,50,51,52,53,54,55,56,57]),
NumValue is Value - 48,
write(NumValue)), !;
(write(' ')).

read_sudoku([
AA, AB, AC, AD, AE, AF, AG, AH, AI,
BA, BB, BC, BD, BE, BF, BG, BH, BI,
CA, CB, CC, CD, CE, CF, CG, CH, CI,
DA, DB, DC, DD, DE, DF, DG, DH, DI,
EA, EB, EC, ED, EE, EF, EG, EH, EI,
FA, FB, FC, FD, FE, FF, FG, FH, FI,
GA, GB, GC, GD, GE, GF, GG, GH, GI,
HA, HB, HC, HD, HE, HF, HG, HH, HI,
IA, IB, IC, ID, IE, IF, IG, IH, II]) :-
write('+---+---+---+'), nl,
write('|'), get_num(AA), get_num(AB), get_num(AC), write('|'),get_num(AD),
   get_num(AE), get_num(AF), write('|'),get_num(AG), get_num(AH),
   get_num(AI), write('|'),nl,
write('|'), get_num(BA), get_num(BB), get_num(BC), write('|'),get_num(BD),
   get_num(BE), get_num(BF), write('|'),get_num(BG), get_num(BH),
   get_num(BI), write('|'),nl,
write('|'), get_num(CA), get_num(CB), get_num(CC), write('|'),get_num(CD),
   get_num(CE), get_num(CF), write('|'),get_num(CG), get_num(CH),
   get_num(CI), write('|'),nl,
write('+---+---+---+'), nl,
write('|'), get_num(DA), get_num(DB), get_num(DC), write('|'),get_num(DD),
   get_num(DE), get_num(DF), write('|'),get_num(DG), get_num(DH),
   get_num(DI), write('|'),nl,
write('|'), get_num(EA), get_num(EB), get_num(EC), write('|'),get_num(ED),
   get_num(EE), get_num(EF), write('|'),get_num(EG), get_num(EH),
   get_num(EI), write('|'),nl,
write('|'), get_num(FA), get_num(FB), get_num(FC), write('|'),get_num(FD),
   get_num(FE), get_num(FF), write('|'),get_num(FG), get_num(FH),
   get_num(FI), write('|'),nl,
write('+---+---+---+'), nl,
write('|'), get_num(GA), get_num(GB), get_num(GC), write('|'),get_num(GD),
   get_num(GE), get_num(GF), write('|'),get_num(GG), get_num(GH),
   get_num(GI), write('|'),nl,
write('|'), get_num(HA), get_num(HB), get_num(HC), write('|'),get_num(HD),
   get_num(HE), get_num(HF), write('|'),get_num(HG), get_num(HH),
   get_num(HI), write('|'),nl,
write('|'), get_num(IA), get_num(IB), get_num(IC), write('|'),get_num(ID),
   get_num(IE), get_num(IF), write('|'),get_num(IG), get_num(IH),
```

```
   get_num(II), write('|'),nl,
write('+---+---+---+'), nl.
```