Cvičení 12

12.1 Opakování

Příklad 12.1.1 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 12.1.2 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 12.1.3 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 12.1.4 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 12.1.5 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 `12\_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 12.1.6 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 12.1.7 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.

12.2 Logické programování s omezujícími podmínkami

Příklad 12.2.1 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 12.2.2 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 12.2.3 Vyřešte pomocí Prologu následující algebrogram:

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

Příklad 12.2.4 Napište predikát `fact/2`, který počítá faktoriál pomocí CLP. V čem je lepší než klasicky implementovaný faktoriál?

Příklad 12.2.5 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 12.2.6 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 12.2.7 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 12.2.8 Stáhněte si program `12\_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.

12.3 Bilingvální opakování

Příklad 12.3.1 V jazyce Haskell naprogramujte rekurzivní funkci `app`, která se chová jako knihovní funkce `++`. Dále v jazyce Prolog naprogramujte odpovídající rekurzivní predikát `app/3`, který se chová jako knihovní predikát `append/3`.

V jazyce Haskell naprogramujte funkci `concat'`, která se chová jako knihovní funkce `concat`. Ekvivalentní predikát `concat/2` napište i v jazyce Prolog. *Pro pokročilejší:* V obou implementacích `concat` zkuste použít tail-rekurzi.

Příklad 12.3.2 V jazyce Haskell naprogramujte funkci `listAvg`, která pro neprázdný seznam čísel vrátí jeho aritmetický průměr. V jazyce Prolog naprogramujte odpovídající predikát `listAvg/2`.

Pro pokročilejší: Zkuste v obou jazycích `listAvg` implementovat tak, aby došlo jen k jednomu průchodu seznamu. V jazyce Haskell můžete zkusit použít vhodný `fold`; v jazyce Prolog akumulátory.

Příklad 12.3.3 V jazyce Haskell naprogramujte funkci `removeDups`, která všechny opakované výskyty libovolného prvku bezprostředně po sobě nahradí pouze jedním výskytem. Například

```
removeDups [1,1,1,2,2,3,1,1,4,4] = [1,2,3,1,4]
```

V jazyce Prolog naprogramujte odpovídající predikát `removeDups/2`.

Příklad 12.3.4 V jazyce Haskell naprogramujte funkci `mergeSort`, která seřadí zadaný seznam pomocí algoritmu *Merge sort*. Může se vám hodit nejprve implementovat pomocné funkce `split` a `merge`.

V jazyce Prolog naprogramujte odpovídající predikát `mergeSort/2`. Zkuste vhodným způsobem použít řezy.

Příklad 12.3.5 V jazyce Haskell naprogramujte funkci `primeFactors`, která pro zadané číslo vrátí seznam čísel, která odpovídají jeho prvočíselnému rozkladu.

Například

```
primeFactors 2 = [2]
primeFactors 4 = [2,2]
primeFactors 6 = [2,3]
primeFactors 30 = [2,3,5]
primeFactors 40 = [2,2,2,5]
```

V jazyce Prolog naprogramujte odpovídající predikát `primeFactors/2`.

Příklad 12.3.6 V jazyce Haskell naprogramujte funkci `quickSort`, která seřadí zadaný seznam pomocí algoritmu *Quick sort*.

V jazyce Prolog naprogramujte odpovídající predikát `quickSort/2`. Zkuste vhodným způsobem použít řezy.

12.4 Opakování Haskell

Příklad 12.4.1 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 12.4.2 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 12.4.3 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 12.4.4 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 12.4.5 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 12.4.6 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 12.4.7 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 12.4.8 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 12.4.9 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" ⇝
    (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 12.4.10 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 12.4.11 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 12.4.12 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 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 12.4.13 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 12.4.14 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])` ≡ `x : reverse s`
b) `map f . filter p` ≡ `filter p . map f`
c) `flip . flip` ≡ `id`
d) `foldr f (foldr f z s) t` ≡ `foldr f z (s ++ t)`
e) `sum (zipWith (+) m n)` ≡ `sum m + sum n`
f) `head s : tail s` ≡ `s`
g) `map f (iterate f x)` ≡ `iterate f (f x)`
h) `foldr (1+) 0` ≡ `length`

Příklad 12.4.15 V následujících výrazech doplňte vhodný podvýraz za `...` tak, aby ekvivalence platily *obecně* (tedy pro libovolnou volbu proměnných).

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

Příklad 12.4.16 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`)

Řešení

Řešení 12.1.1

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 12.1.2 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 12.1.3

```
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 12.1.4

```
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 12.1.5 Řešení zadaných úloh najdete v souboru *einsteinSol.pl*. Řešení využívající jiné kódování najdete v souboru `12\_einsteinSol2.pl`. Soubory naleznete v ISu a v příloze sbírky.

Řešení 12.1.6

```
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 12.1.7 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.
```

Řešení 12.2.1 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 12.2.2 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 12.2.3

```
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 12.2.4

```
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 12.2.5 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 12.2.6

```
:- 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 12.2.7

```
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 12.4.1

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` $\sim$ `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]` $\sim$ `[Bool -> Bool]` a z toho `b` $\sim$ `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` $\sim$ `c -> d`), což je nejobecnější možný typ (unární) funkce.

`f` je však aplikovaná na `42`, z čehož dostáváme unifikaci
`c` $\sim$ `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` $\sim$ `Num c => c`
a typ podvýrazu
`x + x :: Num c => c`

Vidíme, že `f` je funkce, tedy zkonkrétníme typ:
`a` $\sim$ `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` $\sim$ `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` $\sim$ `Num c => c`
`Ord d => d` $\sim$ `f`
Dostáváme tedy `d` $\sim$ `f` $\sim$ `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` $\sim$ `Ord e => e` a
`(> 2) :: (Ord d, Num d) => d -> Bool`.

Dále částečnou aplikací `filter` na `(> 2)` dostáváme unifikaci `b -> Bool` $\sim$ `(Ord d, Num d) => d -> Bool`, a tedy `b` $\sim$ `(Ord d, Num d) => d` a
`filter (> 2) :: (Ord d, Num d) => [d] -> [d]`.

Nyní aplikujeme tuto funkci na argument `xs`, kde dostaneme
`a` $\sim$ `(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` $\sim$ `b -> c`, a tedy `f :: b -> c`, `map f :: [b] -> [c]`.

Tuto funkci dále aplikujeme na `[1,2,3]`: `[b]` $\sim$ `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 12.4.2

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 12.4.3

```
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 12.4.4

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 12.4.5

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` ∼ `Integer`).

Řešení 12.4.6

a) Nekorektní, funkci `max` možno aplikovat maximálně na dva argumenty.

b) Korektní, `(False < True) || True` ⇝ `True || True` ⇝ `True`. `Bool` je instancí typové třídy `Ord`, a tedy hodnoty tohoto typu lze porovnávat.

c) Nekorektní. Sice `5.1 ^ (3.2 ^ 2)` ⇝ `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 případě 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 12.4.7

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 12.4.8

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 12.4.9

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 12.4.10

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 12.4.11 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 12.4.12

```
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 12.4.13

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 12.4.14

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 12.4.15

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 12.4.16

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.