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 Jaký nejobecnější typ může funkce `f` mít, aby byla funkce `g` korektně otypovatelná?

```
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`

Nezapomeňte, že funkce je nutno otypovat čerstvými typovými proměnnými.

2.2 λ-abstrakce

Příklad 2.2.1 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 2.2.2 Jsou následující úpravy korektní?

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

Příklad 2.2.3 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 2.2.4 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) `(\x y -> x + y) 4 0.3`
b) `(\n -> n * 10) (3 + 1)`
c) `(\t -> map t [1, 2, 3]) (+)`
d) `\t u -> t && u 2 3`
e) `\t u -> (t && u 2) 3`
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)`

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 funkci `sums :: [[Int]] -> [Int]`, která ze seznamu seznamů čísel získá seznam součtů vnitřních seznamů. Funkci zadefinujte bez použití knihovních funkcí `map` a `sum`. Příklad použití funkce:

`sums [[1,2,3], [0,1,0], [100], []]` $\leadsto^*$ `[6, 1, 100, 0]`

Příklad 2.3.13 Definujte rekurzivní funkci `applyToList :: (a -> b) -> [a] -> [b]`, která vezme funkci a seznam, a aplikuje danou funkci na každý prvek seznamu.

Příklad 2.3.14 Definujte funkce `add1` a `multiplyN` znovu a co nejkratším zápisem pomocí funkce `applyToList`.

Příklad 2.3.15 Definujte funkci `evens :: [Integer] -> [Integer]`, která ze seznamu čísel vybere jenom sudá.

Příklad 2.3.16 Definujte funkci `evens :: [Integer] -> [Integer]`, která ze seznamu vybere sudá čísla. Použijte funkci `filter`.

Příklad 2.3.17 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"` $\leadsto^*$ `"BOB"`.

Příklad 2.3.18 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]` $\leadsto^*$ `[4,8]`, `multiplyEven [6,6,3]` $\leadsto^*$ `[12,12]`.

Příklad 2.3.19 Definujte funkci `sqroots :: [Double] -> [Double]`, která ze zadaného seznamu vybere kladná čísla a ta odmocní. (Využijte mimo jiné funkce `(>0)` a `sqrt`.)

Příklad 2.3.20 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 2.3.21 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 2.3.22 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 2.3.23 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 2.3.24 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 2.3.25 Napište funkci `brackets`, která vezme řetězec složený ze znaků `'('` a `')'` a rozhodne, jestli se jedná o korektní uzávorkování.

Příklad 2.3.26 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), (5,5), (2,5), (5,1), (2,3)]` $\leadsto$ `[(1,2), (2,5), (5,1)]`

Bonus: Jakým způsobem by šlo najít optimální řešení využívající co nejvíce kostek?

Příklad 2.3.27 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.

Řešení

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

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 2.2.2

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

Řešení 2.2.4

a) `4 + 0.3`
b) `(3 + 1) * 10`
Myšlenkou je, že netřeba zapomenout na zachování závorek po dosazení.
c) `map (+) [1, 2, 3]`
d) `\t u -> t && u 2 3`
V tomto případě není co aplikovat, protože celý zbytek výrazu za šipkou je součástí těla λ-abstrakce.
e) Výraz není korektní. Podvýraz `(t && u 2)` se vyhodnotí na hodnotu typu `Bool` a tu bychom následně aplikovali na číslo `3`, což samozřejmě nelze.
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 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:[])) $\leadsto$ 1 * 2 : multiplyN 2 (2:[])
$\leadsto$ 1 * 2 : (2 * 2 : multiplyN 2 []) $\leadsto$ 1 * 2 : (2 * 2 : [])
$\leadsto^*$ 2 : (4 : []) $\equiv$ [2, 4]

Řešení 2.3.12

```
sums :: [[Int]] -> [Int]
sums [] = []
sums (x:xs) = singleSum x : sums xs
  where singleSum [] = 0
        singleSum (x:xs) = x + singleSum xs
```

Možné je i řešení bez samostatné funkce pro zpracování vnitřních seznamů (i když je trochu méně přehledné).

```
sums' :: [[Int]] -> [Int]
sums' [] = []
sums' ([]:xs) = 0 : sums' xs
sums' ([y]:xs) = y : sums' xs
sums' ((y1:y2:ys):xs) =  sums' ((y1+y2 : ys) : xs)
```

Řešení 2.3.13 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 2.3.14

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

Řešení 2.3.15

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

Řešení 2.3.16

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

Řešení 2.3.17

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

Řešení 2.3.18

```
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 2.3.19

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

Řešení 2.3.21 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 2.3.22 Využívajíc knihovní funkce `map` je řešení velmi krátké.

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

Řešení 2.3.23 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 2.3.24 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 2.3.25

```
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 2.3.26 Zadání je poměrně volné a umožňuje mnoho řešení, dokonce i triviální řešení `domino _ = []`. Užitečnější řešení může fungovat takto: Ze seznamu nejprve vybereme první kostku. Pak opakovaně ve zbytku seznamu najdeme první kostku, která bez otáčení sedí k aktuálnímu konci řetězce, a ostatní nepoužitelné kostky před ní zahazujeme:

```
domino :: (Eq a) => [(a,a)] -> [(a,a)]
domino ((x,y):(z,w):s) = if y == z then (x,y) : domino ((z,w):s)
                                   else domino ((x,y):s)
domino s = s
```

Bonus: Algoritmicky lze tuto úlohu přeložit do řeči teorie grafů jako problém nalezení nejdelší eulerovské cesty v pseudografu (graf s vícenásobnými hranami a smyčkami), kde vrcholy odpovídají číslům na kostkách a hrany kostkám.

Řešení 2.3.27

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