Cvičení 2: Rekurze, seznamy, anonymní funkce

Před druhým cvičením je zapotřebí znát:

- zápis funkce definované pomocí vzorů;
- zápis seznamů pomocí výčtu prvků, tj. například `[1, 2, 10]`;
- základní funkce pro práci se seznamy, tj. například

```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
```

- základní vzory pro seznamy, tj. například `[]`, `(x : xs)`, `[x]`, `[x, y]`;
- lokální definice pomocných funkcí pomocí klauzule `where`;
- chování funkcí

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

- zápis anonymních funkcí pomocí λ-abstrakce.

2.1 Rekurze

Př. 2.1.1 »= Bez použití knihovních funkcí `mod`, `even` a `odd` naprogramujte rekurzivně funkci `isEven`, která pro dané nezáporné celé číslo rozhodne, jestli je sudé. Například:

```
isEven 0 ⇝* True
isEven 3 ⇝* False
isEven 8 ⇝* True
```

Př. 2.1.2 »= Bez použití knihovní funkce `mod` naprogramujte rekurzivně funkci `mod3`, která pro dané nezáporné celé číslo vypočítá jeho zbytek po dělení 3. Například:

```
mod3 0 ⇝* 0
mod3 5 ⇝* 2
mod3 7 ⇝* 1
mod3 9 ⇝* 0
```

Př. 2.1.3 »= Bez použití knihovní funkce `div` naprogramujte rekurzivně funkci `div3`, která dané nezáporné celé číslo celočíselně vydělí třemi. Například:

```
div3 0 ⇝* 0
div3 5 ⇝* 1
div3 7 ⇝* 2
div3 9 ⇝* 3
```

Př. 2.1.4 Definujte rekurzivní funkci pro výpočet faktoriálu.

Př. 2.1.5 Definujte funkci `power` takovou, že `power z n` se pro nezáporné číslo `n` vyhodnotí na `z` na `n`-tou. Například:

```
power 3 1 ⇝* 3
power 3 2 ⇝* 9
power 3 3 ⇝* 27
power 3 0 ⇝* 1
power 0 3 ⇝* 0
```

Př. 2.1.6 Napište funkci `isPower2 :: Integer -> Bool`, která o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky. Můžete použít funkce `even` a `odd`. Například:

```
isPower2 0 ⇝* False
isPower2 1 ⇝* True
isPower2 2 ⇝* True
isPower2 3 ⇝* False
isPower2 4 ⇝* True
isPower2 6 ⇝* False
isPower2 8 ⇝* True
```

Př. 2.1.7 Definujte funkci `digits`, která po aplikaci na kladné celé číslo vrátí jeho ciferný součet. Můžete použít funkce `div` a `mod`. Například:

```
digits 123 ⇝* 6
digits 103 ⇝* 4
```

Př. 2.1.8 Napište funkci `mygcd`, která po aplikaci na dvě kladná celá čísla vrátí jejich největšího společného dělitele. Pokuste se o co nejefektivnější implementaci.

Př. 2.1.9 Naprogramujte funkci `primeDivisors :: Integer -> Integer`, která rozhodne, součinem kolika prvočísel je zadané kladné číslo. Můžete použít funkce `div` a `mod`. Například:


```
primeDivisors 3  ⇝* 1        primeDivisors 8  ⇝* 3
primeDivisors 4  ⇝* 2        primeDivisors 12 ⇝* 3
primeDivisors 5  ⇝* 1        primeDivisors 15 ⇝* 2
primeDivisors 6  ⇝* 2        primeDivisors 1  ⇝* 0
```

Př. 2.1.10 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.

Př. 2.1.11 Co počítá následující funkce? Jak se chová na argumentech, kterými jsou nezáporná čísla? Jak se chová na záporných argumentech?

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

2.2 Seznamy

Př. 2.2.1 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) `[[], [True]]`

Př. 2.2.2 Napište funkci, která na začátek zadaného seznamu čísel vloží číslo `42`. Jaký má vaše funkce typ?

Př. 2.2.3 Bez použití jakýchkoli knihovních funkcí definujte funkci `isEmpty` typu `[a] -> Bool`, která pro prázdný vstupní seznam vrátí `True` a pro neprázdný vrátí `False`.

Př. 2.2.4 Bez použití funkcí `head` a `tail` napište funkci `myHead` typu `[a] -> a`, která vrátí první prvek zadaného neprázdného seznamu, a funkci `myTail` typu `[a] -> [a]`, která pro zadaný neprázdný seznam vrátí tentýž seznam bez prvního prvku.

Př. 2.2.5 Definujte funkci `second :: [a] -> a`, která vrátí druhý prvek zadaného seznamu. Můžete předpokládat, že vstupní seznam obsahuje alespoň dvě čísla.

Př. 2.2.6 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]]`

Př. 2.2.7 Bez použití knihovní funkce `last` definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu.

Př. 2.2.8 Bez použití funkce `init` definujte funkci `stripLast` typu `[a] -> [a]`, která pro neprázdný seznam vrátí tentýž seznam bez posledního prvku.

Př. 2.2.9 Bez použití knihovní funkce `length` definujte funkci `len :: [a] -> Integer`, která spočítá délku zadaného seznamu.

Př. 2.2.10 Napište funkci `containsNumber :: Integer -> [Integer] -> Bool`, která vrací `True`, pokud je první argument obsažen v seznamu zadaném druhým argumentem, jinak vrací `False`. Například:

```
containsNumber 42 [1, 2, 3] ⇝* False
containsNumber 2 [1, 2, 3] ⇝* True
containsNumber 2 [] ⇝* False
```

Př. 2.2.11 Napište funkci `containsNNumbers` typu `Integer -> Integer -> [Integer] -> Bool` takovou, že `containsNNumbers n x xs` bude `True` právě tehdy, když seznam `xs` obsahuje alespoň `n` výskytů čísla `x`. Například:

```
containsNNumbers 2 42 [1, 2, 42] ⇝* False
containsNNumbers 2 42 [1, 42, 2, 42] ⇝* True
containsNNumbers 3 42 [1, 42, 2, 42] ⇝* False
containsNNumbers 0 42 [1, 2] ⇝* True
```

Př. 2.2.12 Mějme neprázdný seznam typu `[(String, Integer)]`, který reprezentuje seznam jmen studentů s jejich počty bodů z předmětu IB015. Naprogramujte

a) funkci `getPoints :: String -> [(String, Integer)] -> Integer`, která vrátí počet bodů studenta se jménem zadaným v prvním argumentu (nebo 0, pokud takový student v seznamu není),

b) funkci `getBest :: [(String, Integer)] -> String`, která vrátí jméno studenta s nejvíce body.

Například tedy:

```
getPoints "Stan" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* 20
getPoints "Tomas" [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* 0
getBest [("Kyle", 30), ("Eric", 42), ("Stan", 20)] ⇝* "Eric"
```

Př. 2.2.13 Napište funkci `nth :: Integer -> [a] -> a`, která ze zadaného seznamu vrátí prvek na pozici určené prvním argumentem funkce. Můžete předpokládat, že vstupní seznam má dostatečnou délku. Například:

```
nth 0 [4, 3, 5] ⇝* 4
nth 1 [4, 3, 5] ⇝* 3
nth 2 [4, 3, 5] ⇝* 5
nth 2 "Get Schwifty" ⇝* 't'
```

Př. 2.2.14 Napište funkci `append :: [a] -> [a] -> [a]`, jejíž výsledek pro dva seznamy bude seznam, který vznikne zřetězením těchto dvou seznamů. Nepoužívejte knihovní operátor `(++)`. Například:

```
append [1, 2] [3, 4, 8] ⇝* [1, 2, 3, 4, 8]
append [] [3, 4] ⇝* [3, 4]
append [3, 4] [] ⇝* [3, 4]
append "Legen" "dary" ⇝* "Legendary"
```

Př. 2.2.15 Napište funkci `pairs :: [a] -> [(a, a)]` , která bere prvky ze vstupního seznamu po dvou prvcích a vytváří seznam dvojic těchto prvků. Pokud má seznam lichý počet prvků, poslední prvek se zahodí. Například:

```
pairs [4, 8, 15, 16, 23] ⇝* [(4, 8), (15, 16)]
pairs [4, 8, 15, 16, 23, 42] ⇝* [(4, 8), (15, 16), (23, 42)]
pairs "Humphrey" ⇝* [('H', 'u'), ('m', 'p'), ('h', 'r'), ('e', 'y')]
```

Př. 2.2.16 Napište následující funkce pracující se seznamy čísel pomocí rekurze a vzorů:

a) `listSum :: [Integer] -> Integer`, která dostane seznam čísel a vrátí součet všech jeho prvků.

b) `oddLength :: [Integer] -> Bool`, která vrátí `True`, pokud je seznam liché délky, jinak `False` (bez použití funkce `length`).

c) `add1 :: [Integer] -> [Integer]`, která každé číslo ve vstupním seznamu zvýší o 1,

d) `multiplyN :: Integer -> [Integer] -> [Integer]`, která každé číslo ve vstupním seznamu vynásobí prvním argumentem funkce,

e) `deleteEven :: [Integer] -> [Integer]`, která ze seznamu čísel odstraní všechna sudá čísla,
f) `deleteElem :: Integer -> [Integer] -> [Integer]`, která ze seznamu čísel odstraní všechny výskyty čísla zadaného prvním argumentem,
g) `largestNumber :: [Integer] -> Integer`, vrátí největší číslo ze zadaného neprázdného seznamu čísel,
h) `listsEqual :: [Integer] -> [Integer] -> Bool`, která dostane na vstup dva seznamy a vrátí `True` právě tehdy, když se rovnají (bez použití funkce `(==)` na seznamy),
i) `multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2 (lichá čísla vynechá),
j) `sqroots :: [Double] -> [Double]`, která ze zadaného seznamu vybere kladná čísla a ta odmocní (může se vám hodit funkce `sqrt`).

Př. 2.2.17 Napište funkci `everyNth :: Integer -> [a] -> [a]` takovou, že seznam `everyNth n xs` bude obsahovat každý n-tý prvek ze seznamu `xs`. Například:


```
everyNth 2 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 1, 2, 7]
everyNth 3 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 3, 7]
everyNth 4 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 2]
everyNth 1 [6, 8, 1, 3, 2, 5, 7] ⇝* [6, 8, 1, 3, 2, 5, 7]
everyNth 2 "BoJack Horseman" ⇝* "BJc osmn"
```

Př. 2.2.18 Napište funkci `brackets :: String -> Bool`, která dostane řetězec složený ze znaků `'('` a `')'` a rozhodne, jestli se jedná o korektní uzávorkování. Například:


```
brackets "(()())" ⇝* True
brackets "(()()" ⇝* False
brackets "())()" ⇝* False
brackets "())(" ⇝* False
brackets "" ⇝* True
```

Př. 2.2.19 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. Tedy například:


```
palindrome "brienne" ⇝* False
palindromize "brienne" ⇝* "brienneirb"
```

Př. 2.2.20 Napište funkci `getMiddle :: [a] -> a`, která pro zadaný neprázdný seznam vrátí jeho prostřední prvek bez zjišťování jeho délky. Pokud má seznam sudý počet prvků, vraťte levý z prostředních dvou. Například:

```
getMiddle [1] ⇝* 1
getMiddle [2, 1] ⇝* 2
getMiddle [2, 1, 5] ⇝* 1
getMiddle [2, 1, 5, 6] ⇝* 1
getMiddle [2, 1, 5, 6, 3] ⇝* 5
getMiddle "Don't blink!" ⇝* ' '
```

Nápověda: Pokud zajíc běží dvakrát rychleji než želva, pak v okamžiku, kdy zajíc vyhrál závod, je želva v polovině trati.

2.3 Užitečné seznamové funkce

> **Pan Fešák doporučuje:** Pokud si nejste chováním některé funkce jistí, můžete ji najít v **dokumentaci**. Teď se například může hodit najít si seznamové funkce jako `map` a `filter`.

Př. 2.3.1 Slovně vysvětlete, co dělají knihovní funkce `map` a `filter`. Pro jaký účel byste použili kterou z nich? Jaký je rozdíl mezi výsledkem vyhodnocení `map even [1, 3, 2, 5, 4, 8, 11]` a `filter even [1, 3, 2, 5, 4, 8, 11]`?

Př. 2.3.2 Mějme seznam typu `[(String, Integer)]`, který obsahuje jména studentů a jejich počty bodů z předmětu IB015. Pomocí vhodných seznamových funkcí naprogramujte

a) funkci `getNames :: [(String, Integer)] -> [String]`, která vrátí seznam jmen studentů,

b) funkci `successfulRecords :: [(String, Integer)] -> [(String, Integer)]`, která ze zadaného seznamu vybere záznamy těch studentů, kteří mají alespoň 50 bodů,

c) funkci `successfulNames :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere jména studentů, kteří mají alespoň 50 bodů,

d) funkci `successfulStrings :: [(String, Integer)] -> [String]`, která ze zadaného seznamu vybere studenty, kteří mají alespoň 50 bodů, a vrátí seznam řetězců ve tvaru `"jmeno: xxx b"` (*nápověda*: pro převod čísla na řetězec můžete použít funkci `show`).

Tedy například pro databázi

```
st :: [(String, Integer)]
st = [("Finn", 35), ("Jake", 53), ("Bubblegum",  98),
      ("Ice King", 20), ("BMO", 99)]
```

budou požadované funkce vracet následující hodnoty:

```
getNames st ⇝* ["Finn", "Jake", "Bubblegum", "Ice King", "BMO"]
successfulRecords st ⇝* [("Jake", 53), ("Bubblegum",  98), ("BMO", 99)]
successfulNames st ⇝* ["Jake", "Bubblegum", "BMO"]
successfulStrings st ⇝* ["Jake: 53 b", "Bubblegum: 98 b", "BMO: 99 b"]
```

Př. 2.3.3 Které z funkcí z příkladu 2.2.16 lze elegantně naprogramovat pomocí funkce `map`? Které lze elegantně naprogramovat pomocí funkce `filter`? Všechny tyto funkce pomocí `map` a `filter` naprogramujte.

Př. 2.3.4 S využitím funkce `map` a knihovní funkce `toUpper :: Char -> Char` z modulu `Data.Char` (tj. je třeba použít `import Data.Char`, na začátku souboru, nebo `:m +Data.Char` v interpretu) definujte novou funkci `toUpperStr`, která převádí řetězec písmen na řetězec velkých písmen. Například:

```
toUpperStr "i am the one who knocks!" ⇝* "I AM THE ONE WHO KNOCKS!"
```

Př. 2.3.5 Napište funkci `vowels`, která dostane seznam řetězců a vrátí seznam řetězců takových, že v každém řetězci ponechá jenom samohlásky (ale zachová jejich pořadí). Například:

```
vowels ["Michael", "Dwight", "Jim", "Pam"] ⇝* ["iae", "i", "i", "a"]
vowels ["MICHAEL", "DWIGHT", "JIM", "PAM"] ⇝* ["IAE", "I", "I", "A"]
```

Př. 2.3.6 Slovně vysvětlete, co dělají funkce `zip` a `zipWith`. Pro jaký účel byste použili kterou z nich? Pomocí interpretu zjistěte, jak se tyto funkce chovají, pokud mají vstupní seznamy různou délku.

Př. 2.3.7 Mějme výsledky běžeckého závodu reprezentované pomocí seznamu typu `[String]`, který obsahuje jména běžců seřazených od nejlepšího po nejhoršího, a seznam peněžních výher typu `[Integer]`. Naprogramujte

a) funkci `assignPrizes` typu `[String] -> [Integer] -> [(String, Integer)]`, která každému běžci, který něco vyhrál, přiřadí jeho výhru, a
b) funkci `prizeTexts` typu `[String] -> [Integer] -> [String]`, která vrátí seznam řetězců ve tvaru `"jmeno: xxx Kc"` pro každého běžce, který něco vyhrál.

Například:

```
assignPrizes ["Mike", "Dustin", "Lucas", "Will"] [100, 50] ⇝*
    [("Mike", 100), ("Dustin", 50)]
prizeTexts ["Mike", "Dustin", "Lucas", "Will"] [100, 50] ⇝*
    ["Mike: 100 Kc", "Dustin: 50 Kc"]
```

Př. 2.3.8 Pomocí funkce `zip` napište funkci `neighbors :: [a] -> [(a, a)]`, která pro zadaný seznam vrátí seznam dvojic sousedních prvků. Například:

```
neighbors [3, 8, 2, 5] ⇝* [(3, 8), (8, 2), (2, 5)]
neighbors [3, 8] ⇝* [(3, 8)]
neighbors [3] ⇝* []
neighbors "Kree!" ⇝* [('K', 'r'), ('r', 'e'), ('e', 'e'), ('e', '!')]
```

Př. 2.3.9 Napište funkci, která zjistí, jestli jsou v seznamu čísel některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

Př. 2.3.10 Implementujte funkce `myMap`, `myFilter` a `myZipWith`, které se budou chovat jako knihovní funkce `map`, `filter` a `zipWith`.

Př. 2.3.11 Uvažte funkci `anyEven :: [Integer] -> Bool`, která rozhodne, jestli je v seznamu čísel nějaké kladné číslo, a funkci `allEven :: [Integer] -> Bool`, která rozhodne, jestli jsou všechna čísla v seznamu kladná. Najděte ve standardní knihovně funkci nebo funkce, pomocí kterých lze funkce `anyEven` a `allEven` implementovat jednoduše bez explicitního použití rekurze.

Př. 2.3.12 Zjistěte, co dělají funkce `takeWhile` a `dropWhile`.

2.4 λ-abstrakce

Př. 2.4.1 Slovně popište, co dělají následující funkce:

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

Př. 2.4.2 Implementujte funkce z příkladu 2.3.2 tak, že místo vlastních pomocných funkcí použijete λ-abstrakci.

Př. 2.4.3 Implementujte všechny funkce ze sekce 2.3 tak, že místo vlastních pomocných funkcí použijete λ-abstrakci.

Př. 2.4.4 
Pomocí rekurze a funkce `filter` napište funkci `quickSort :: [Integer] -> [Integer]`, která seřadí vstupní seznam vzestupně pomocí algoritmu *quick sort*. Například:

```
quickSort [5, 3, 8, 12, 1] ⇝* [1, 3, 5, 8, 12]
quickSort [5, 4, 3, 2] ⇝* [2, 3, 4, 5]
quickSort [2, 2, 2] ⇝* [2, 2, 2]
quickSort [2] ⇝* [2]
quickSort [] ⇝* []
```

Pokud algoritmus quick sort neznáte, zkuste ho nastudovat například na Wikipedii.

Na konci druhého cvičení byste měli umět:

- definovat vlastní rekurzivní funkce pracující s celými čísly;
- pracovat se seznamy a definovat na nich funkce pomocí vzorů;
- definovat rekurzivní funkce na seznamech;
- poznat, kdy je na práci se seznamy vhodné použít knihovní funkce `map`, `filter`, `zip` a `zipWith`, a umět tyto funkce použít;
- poznat, kdy je vhodné použít λ-abstrakci, a umět pomocí λ-abstrakce definovat jednoduché funkce.

Řešení

Řeš. 2.1.1 Myšlenka řešení může být následující: 0 je sudá, 1 není sudá, a každé jiné kladné číslo je sudé právě tehdy, když číslo o 2 menší je sudé.

```
isEven :: Integer -> Bool
isEven 0 = True
isEven 1 = False
isEven x = isEven (x - 2)
```

Řeš. 2.1.2 Myšlenka řešení může být následující: zbytek 0 po dělení třemi je 0, zbytek 1 po dělení třemi je 1, zbytek 2 po dělení třemi je 2 a zbytek dělení třemi pro každé jiné kladné číslo je stejný, jako zbytek po dělení třemi pro číslo o 3 menší.

```
mod3 :: Integer -> Integer
mod3 0 = 0
mod3 1 = 1
mod3 2 = 2
mod3 x = mod3 (x - 3)
```

Řeš. 2.1.3 Myšlenka je podobná jako v předešlém příkladu. Jen se zde počítá, kolikrát je potřeba odečíst 3, aby se argument dostal do intervalu $\langle 0, 3\rangle$.

```
div3 :: Integer -> Integer
div3 0 = 0
div3 1 = 0
div3 2 = 0
div3 x = 1 + div3 (x - 3)
```

Řeš. 2.1.4

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

Funkce je opět definována po částech. Předpokládáme, že dostane jako argument jenom nezáporné celé číslo. Pokud je argument 0, výsledek je zřejmě opět 0. Pokud je naopak argument kladné číslo, víme, že $n! = n \times (n-1)!$, kde druhý výsledek získáme pomocí rekurzivního volání. Poznamenejme, že závorky kolem `n - 1` je nutno použít, protože jinak by se výraz implicitně uzávorkoval jako `(fact n) - 1`, protože aplikace prefixově zapsané funkce má vyšší prioritu než infixové operátory.

Řeš. 2.1.5 Bázový případ $z^0 = 1$. Pro každé kladné n naopak platí $z^n = z \cdot z^{n-1}$, přičemž hodnota z^{n-1} jde vypočítat rekurzivně.

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

Všimněte si, že definovaná funkce vždy použije n rekurzivních volání. Tento počet se ale dá zmenšit, když si uvědomíme, že při každém rekurzivním volání není potřeba n snižovat jen o 1, ale je možné ho zmenšit přibližně na polovinu. Platí totiž

- $z^0 = 1$
- $z^{2n} = (z^n)^2$

- $z^{2n+1} = (z^n)^2 \cdot z$

Následující implementace, která tuto myšlenku využívá, tedy potřebuje jen přibližně $\log(n)$ rekurzivních volání.

```
power' _ 0 = 1
power' z n = if even n then half * half else half * half * z
    where half = power' z (n `div` 2)
```

Řeš. 2.1.6 Stačí si uvědomit, že číslo je mocninou 2 právě tehdy, když je 1 nebo je sudé a jeho polovina je mocninou 2.

```
isPower2 :: Integer -> Bool
isPower2 0 = False
isPower2 1 = True
isPower2 x = even x && isPower2 (div x 2)
```

Řeš. 2.1.7

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

Řeš. 2.1.8 K řešení můžeme použít známý Euklidův algoritmus. Konkrétně použijeme jeho rekurzivní verzi, která využívá zbytky po dělení.

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

Řeš. 2.1.9 Příklad lze vyřešit i bez generování všech prvočísel. Stačí zadané číslo dělit 2 tolikrát, kolikrát je to možné, a zároveň přičítat za každé vydělení k výsledku jedničku. Poté číslo budeme dělit 3, 4, 5, 6, atd., dokud po dělení nezbyde výsledek 1. Ačkoliv například čísla 4 a 6 nejsou prvočísla, ale přesto jimi dělíme, není to problém, protože pokud jsme číslo už vydělili 2 a 3, kolikrát to bylo možné, nemůže už být dělitelné ani 4 ani 6.

```
primeDivisors :: Integer -> Integer
primeDivisors n = divisorsFrom n 2
    where
        divisorsFrom 1 _ = 0
        divisorsFrom n d = if mod n d == 0
                           then 1 + divisorsFrom (n `div` d) d
                           else divisorsFrom n (d + 1)
```

Řeš. 2.1.10

```
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))
```

Řeš. 2.1.11 Pro zadané číslo n funkce přičte $2n - 1$ k výsledku rekurzivního volání pro vstup o jedna menší. Ten vrátí $2(n-1) - 1$ a součet rekurzivního volání o 1 menší. To se bude dít tak dlouho, než vstup bude 0, pro nějž funkce vrátí 0. Tedy výsledkem bude součet

$$(2n-1) + (2(n-1)-1) + (2(n-2)-1) + \ldots + (2-1) + 0.$$

To lze zapsat přesněji též jako

$$\sum_{i=1}^{n}(2n-1).$$

Předchozí výraz je perfektně správné řešení úlohy. Nicméně s použitím trochy matematiky lze řešení zapsat i elegantněji, aby bylo opravdu vidět, co zadaná funkce počítá.

Odečtení n jedniček lze vytknout za celý součet, a tedy výsledek lze zapsat i jako

$$\left(\sum_{i=1}^{n} 2n\right) - n.$$

Násobení dvojkou lze též vytknout před součet, a tedy výsledek lze zapsat i jako

$$2\left(\sum_{i=1}^{n} n\right) - n.$$

Ze střední školy možná víte, že součet aritmetické řady $\sum_{i=1}^{n} n$ je $\frac{n\cdot(n+1)}{2}$. Takže výsledek lze zapsat též jako

$$2\frac{n\cdot(n+1)}{2} - n = n\cdot(n+1) - n = n\cdot n = n^2.$$

Pokud si chcete procvičit látku z Matematických základů informatiky, můžete si zkusit právě odvozenou rovnost

$$\sum_{i=1}^{n}(2n-1) = n^2$$

dokázat matematickou indukcí.

Řeš. 2.2.1 Použijte příkaz `: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]`, při otypování takovýchto výrazů je třeba si dát pozor. Výraz sice obsahuje funkci `++`, která má v tomto kontextu typ `String -> String -> String`, avšak `String -> String -> String` není výsledný typ, protože funkci už byly dodány 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) `[[Bool]]` typové omezení vzniká kvůli konkrétní hodnotě ve druhém prvku (prázdný řetězec).

Řeš. 2.2.2 Stačí použít funkci `(:)`.

```
add42 :: [Integer] -> [Integer]
add42 xs = 42 : xs
```

Řeš. 2.2.3 Stačí použít definici funkce podle vzorů. Prázdný seznam je prázdný, žádný jiný seznam není prázdný (duh).

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Řeš. 2.2.4 Opět stačí použít definici funkce podle vzorů. Případy, kdy vstupem funkce je prázdný seznam, buď není potřeba vůbec definovat, nebo lze použít funkci `error` nebo hodnotu `undefined`.

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

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

Řeš. 2.2.5 Stačí si vzpomenout na vzor `x : y : xs` pro seznam, který má alespoň dva prvky.

```
second :: [a] -> a
second (_ : y : _) = y
second _           = error "Your list is too short :("
```

Řeš. 2.2.6

- `[]`
  Tento vzor představuje prázdný seznam. Nemůže reprezentovat žádný z uvedených seznamů.
- `x`
  Na tento vzor se může navázat libovolná hodnota, a tedy zejména libovolný ze zadaných seznamů.
- `[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 (ano, i `[[]]`).
- `(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, jehož 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, jehož 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]]`.

Řeš. 2.2.7 Případ prázdného seznamu nemusíme řešit. Pro jednoprvkový seznam vrátíme rovnou jeho poslední prvek:

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

Všechny zbývající případy seznamů mají alespoň dva prvky. Jednoduchá úvaha vede k tomu, že poslední prvek seznamu, který má alespoň dva prvky, je stejný jako poslední prvek téhož seznamu, ale bez prvního prvku. Tedy ze vstupního seznamu odstraníme první prvek a na zbytek aplikujeme rekurzivně funkci `getLast`:

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

Řeš. 2.2.8 Funkci definujeme 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 případě bude první prvek zadaného seznamu určitě ve výsledném seznamu a zbytek lze vypočítat rekurzivně:

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

Srovnejte s definicí funkce `getLast`.

Řeš. 2.2.9 Délka prázdného seznamu je 0. Délka alespoň jednoprvkového seznamu je o 1 větší, než délka vstupního seznamu bez prvního prvku.

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

Výpočet této funkce probíhá například takto:

```
len (1 : (2 : [])) ⇝ 1 + len (2 : []) ⇝ 1 + (1 + len [])
    ⇝ 1 + (1 + 0) ⇝* 2
```

Řeš. 2.2.10 Prázdný seznam neobsahuje žádné číslo. Neprázdný seznam obsahuje zadané číslo právě tehdy, když je ono číslo prvním prvkem zadaného seznamu, nebo ho obsahuje zbytek zadaného seznamu.

```
containsNumber :: Integer -> [Integer] -> Bool
containsNumber _ [] = False
containsNumber e (x : xs) = e == x || containsNumber e xs
```

Řeš. 2.2.11 Myšlenka je podobná jako u funkce `containsNumber`, jen je potřeba si počítat, kolikrát zadané číslo ještě chceme vidět. Pokud se dostaneme na 0, číslo už jsme viděli dostatečněkrát, a vrátíme tedy `True`.

```
containsNNumbers :: Integer -> Integer -> [Integer] -> Bool
containsNNumbers 0 _ _          = True
containsNNumbers _ _ []         = False
containsNNumbers n x (y : ys) = if x == y
    then containsNNumbers (n - 1) x ys
    else containsNNumbers n x ys
```

Řeš. 2.2.12 Funkce `getPoints` je podobná funkci `containsNumber`, ale místo logické hodnoty budeme vracet příslušnou hodnotu.

```
getPoints :: String -> [(String, Integer)] -> Integer
getPoints _ [] = 0
getPoints wanted ((name, points) : xs) = if wanted == name
    then points
    else getPoints wanted xs
```

U funkce `getBest` se hodí definovat si pomocnou funkci, která jako další argument dostane i jméno a počet bodů aktuálně nejlepšího studenta. Tohoto aktuálně nejlepšího studenta bude v průběhu vypočtu měnit a na konci seznamu ho vrátí.

```
getBest :: [(String, Integer)] -> String
getBest (x : xs) = fst (getBestWithDefault x xs)
    where
        getBestWithDefault current [] = current
        getBestWithDefault (curN, curP) ((newN, newP) : xs) =
            if newP > curP
            then getBestWithDefault (newN, newP) xs
            else getBestWithDefault (curN, curP) xs
```

Řeš. 2.2.13

```
nth :: Integer -> [a] -> a
nth 0 (x : _)  = x
nth n (_ : xs) = nth (n - 1) xs
```

Řeš. 2.2.14 Nejjednodušší je funkci definovat podle vzoru na prvním argumentu. Pokud je první seznam prázdný, výsledkem je přímo druhý seznam. Pokud je první seznam neprázdný, výsledný seznam obsahuje první prvek prvního seznamu a pak zřetězení zbytku prvního seznamu s druhým seznamem.

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x : xs) ys = x : append xs ys
```

Řeš. 2.2.15 Zde se hodí vzor pro seznamy délky alespoň dva, protože potřebujeme pojmenovat první dva prvky vstupního seznamu. Poté stačí udělat z nich dvojici a zbytek seznamu vyřešit rekurzivně.

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

Řeš. 2.2.16 a) Součet prázdného seznamu je 0. Součet neprázdného seznamu je součet prvního prvku a součtu zbytku seznamu.

```
listSum :: [Integer] -> Integer
listSum []       = 0
```

```
listSum (x : xs) = x + listSum xs
```

b) Existují nejméně dva přístupy k řešení, pokud nechceme explicitně pracovat s délkou seznamu. Jeden z nich je, že využijeme vzoru `(x : y : zs)`, který bere ze seznamu *po dvou* prvcích, tím pádem víme, že pokud tak skončíme na jednom prvku, seznam musel obsahovat lichý počet prvků:

```
oddLength :: [Integer] -> Bool
oddLength [] = False
oddLength [_] = True
oddLength (_ : _ : zs) = oddLength zs
```

Druhý přístup k řešení je, že odpověď postupně *vyskládáme* z prázdného seznamu, protože víme, že ten obsahuje sudý počet prvků. Každým dalším prvkem odpověď změníme na opačnou, s posledním prvkem získáme odpověď pro celý seznam:

```
oddLength' :: [Integer] -> Bool
oddLength' [] = False
oddLength' (_ : xs) = not (oddLength xs)
```

c) Opět přímočará rekurzivní definice funkce podle vzorů.

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

d) Opět přímočará rekurzivní definice funkce podle vzorů.

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

e) Opět přímočará rekurzivní definice funkce podle vzorů.

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

f) Opět přímočará rekurzivní definice funkce podle vzorů.

```
deleteElem :: Integer -> [Integer] -> [Integer]
deleteElem _ [] = []
deleteElem n (x : xs) = if x == n
    then deleteElem n xs
    else x : deleteElem n xs
```

g) Opět přímočará rekurzivní definice funkce podle vzorů.

```
largestNumber :: [Integer] -> Integer
largestNumber [x] = x
largestNumber (x : xs) = x `max` largestNumber xs
```

h) Stačí si uvědomit, jaké všechny případy mohou nastat. Jediný zajímavý případ je, když oba vstupní seznamy jsou neprázdné. V takovém případě je potřeba porovnat první prvky obou seznamů a také rekurzivně porovnat zbytky obou seznamů.

```
listsEqual :: [Integer] -> [Integer] -> Bool
listsEqual [] [] = True
listsEqual [] _ = False
listsEqual _ [] = False
listsEqual (x : xs) (y : ys) = x == y && listsEqual xs ys
```

i) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
multiplyEven :: [Integer] -> [Integer]
multiplyEven [] = []
multiplyEven (x : xs) = if even x
    then (2 * x) : multiplyEven xs
    else multiplyEven xs
```

j) Tentokrát jen trochu komplikovanější rekurzivní definice funkce podle vzorů.

```
sqroots :: [Double] -> [Double]
sqroots [] = []
sqroots (x : xs) = if x > 0
    then sqrt x : sqroots xs
    else sqroots xs
```

Řeš. 2.2.17

```
everyNth :: Integer -> [a] -> [a]
everyNth n xs = everyNthOffset n xs 0
    where
        everyNthOffset _ [] _ = []
        everyNthOffset n (x : xs) 0 = x : everyNthOffset n xs (n - 1)
        everyNthOffset n (x : xs) m = everyNthOffset n xs (m - 1)
```

Řeš. 2.2.18

```
brackets :: String -> Bool
brackets s = bracketsWithDiff s 0
    where
        bracketsWithDiff [] k = k == 0
        bracketsWithDiff ('(' : xs) k = bracketsWithDiff xs (k + 1)
        bracketsWithDiff (')' : xs) k = k > 0 &&
                                        bracketsWithDiff xs (k - 1)
```

Řeš. 2.2.19 Funkci, která rozhodne, jestli je řetězec palindromem, zadefinujeme jednoduše pomocí funkce `reverse` a porovnání.

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

Po krátkém zamyšlení zjistíme, že na doplnění slova na palindrom nám stačí najít nejdelší příponu slova, která tvoří palindrom. Vynechané znaky ze začátku pak doplníme i 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.

Řeš. 2.2.20

```
getMiddle :: [a] -> a
getMiddle xs = tortoiseRabbit xs xs
    where
        tortoiseRabbit (t : _) [_] = t
        tortoiseRabbit (t : _) [_, _] = t
        tortoiseRabbit (_ : ts) (_ : _ : rs) = tortoiseRabbit ts rs
```

Řeš. 2.3.2

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

successfulRecords :: [(String, Integer)] -> [(String, Integer)]
successfulRecords s = filter successful s
    where
        successful (_, p) = p >= 50

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

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

Řeš. 2.3.3 Funkci `map` lze využít v případě, že potřebujeme jistým způsobem modifikovat každý prvek zadaného seznamu.

```
add1' :: [Integer] -> [Integer]
add1' xs = map plus1 xs
    where plus1 x = x + 1

multiplyN' :: Integer -> [Integer] -> [Integer]
multiplyN' n xs = map timesN xs
    where timesN x = x * n
```

Funkci `filter` naopak použijeme, chceme-li ze vstupního seznamu vybrat pouze některé prvky.

```
deleteEven' :: [Integer] -> [Integer]
-- chci odstranit sudá čísla, ponechám tedy ty prvky,
-- pro které platí odd (číslo je liché)
deleteEven' xs = filter odd xs

deleteElem' :: Integer -> [Integer] -> [Integer]
deleteElem' n xs = filter notEqualN xs
    where notEqualN x = x /= n
```

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

```
multiplyEven' :: [Integer] -> [Integer]
multiplyEven' xs = map times2 (filter even xs)
    where times2 x = x * 2

sqroots' :: [Double] -> [Double]
sqroots' xs = map sqrt (filter greaterThan0 xs)
    where greaterThan0 x = x > 0
```

Zbývající funkce nelze vhodně implementovat pomocí `map` a `filter`: `listSum`, `oddLength` a `listsEqual` se vyhodnocují na jeden prvek (typu `Integer` nebo `Bool`), ale `map` a `filter` vrací seznamy. Funkce `listsEqual` musí najednou procházet dva seznamy, ale `map` a `filter` rekurzivně prochází vždy pouze jeden seznam (lze elegantně řešit použitím funkce `zipWith`, je však potřeba ohlídat, zda mají seznamy stejnou délku).

Řeš. 2.3.4
```
import Data.Char
toUpperStr :: String -> String
toUpperStr = map toUpper
```

Řeš. 2.3.5 Nejdřív si zadefinujeme pomocný predikát `isvowel`, který o znaku určí, jestli je samohláskou. Následně jednotlivé řetězce projdeme funkcí `filter`.
```
isvowel :: Char -> Bool
isvowel c = elem (toUpper c) "AEIOUY"
vowels :: [String] -> [String]
vowels s = map (filter isvowel) s
```

Řeš. 2.3.7
```
assignPrizes :: [String] -> [Integer] -> [(String, Integer)]
assignPrizes = zip

formatPrizeText :: String -> Integer -> String
formatPrizeText n p = n ++ ": " ++ show p ++ " Kc"

prizeTexts :: [String] -> [Integer] -> [String]
prizeTexts ns ps = zipWith formatPrizeText ns ps
```

Řeš. 2.3.8
```
neighbors :: [a] -> [(a, a)]
neighbors xs = zip xs (tail xs)
```

Řeš. 2.3.9
```
f1 :: [Integer] -> Bool
f1 (x : y : s) = x == y || f1 (y : s)
f1 _           = False
```

Nebo kratší řešení používající funkci `zipWith` a funkci `or`, která spočítá logický součet všech hodnot v zadaném seznamu:
```
f2 :: [Integer] -> Bool
f2 s = or (zipWith (==) s (tail s))
```

Řeš. 2.3.10
```
myMap :: (a -> b) -> [a] -> [b]
myMap f [] = []
myMap f (x : xs) = f x : myMap f xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFilter p (x : xs) = if p x
    then x : myFilter p xs
    else myFilter p xs

myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith f (x : xs) (y : ys) = f x y : myZipWith f xs ys
myZipWith _ _ _ = []
```

Řeš. 2.3.11 Jedná se o funkce `any` a `all`.
http://hackage.haskell.org/package/base/docs/Prelude.html#v:any
http://hackage.haskell.org/package/base/docs/Prelude.html#v:all

Řeš. 2.3.12 http://hackage.haskell.org/package/base/docs/Prelude.html#v:takeWhile

Řeš. 2.4.4

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