Cvičení 1: Základní konstrukce

Před cvičením je nezbytné umět:

- otevřít terminál (příkazovou řádku) a vytvořit textový soubor;
- spustit interpret GHCi a načíst do něj soubor;
- používat GHCi jako kalkulačku;
- znát na intuitivní úrovni pojmy *funkce* a *typ*;
- vědět, jakým způsobem se volají funkce v Haskellu;
- vědět, jaké základní typy v Haskellu jsou;
- vědět, jak fungují základní konstrukce `if ... then ... else ...`, `let ... in ...` a `where`;
- umět napsat jednoduchou funkci včetně využití více definičních rovností a vzorů.

Příkazy interpretu GHCi:

`:t[ype]` *`výraz`* – typ výrazu

`:i[nfo]` *`jméno`* – informace o operátorech, funkcích a typech

`:doc` *`jméno`* – dokumentace operátorů, funkcí a typů (až od GHC 8.6)

`:l[oad]` *`soubor.hs`* – načtení souboru s Haskellovým kódem

`:r[eload]` – znovunačtení posledního souboru

`:q[uit]` – ukončení práce s interpretem

`:m[odule]` *`Modul`* – načtení modulu (bude používáno později)

`:h[elp]` – nápověda

Pan Fešák připomíná: Všechny funkce v Haskellu jsou *čisté funkce*, tj. nemají žádný vnitřní stav a pro stejné argumenty vždy vrací stejnou hodnotu (na rozdíl od funkcí v Pythonu).

Etudy

Etuda 1.η.1 S pomocí dokumentace zjistěte, jaký je rozdíl mezi typy `Int` a `Integer`.

Etuda 1.η.2 Definujte funkci `isSucc :: Integer -> Integer -> Bool`, která pro dvě celá čísla `x`, `y` rozhodne, jestli je `y` bezprostředním následníkem `x`.

```
isSucc 1 4 ⇝* False
isSucc 4 5 ⇝* True
```

Etuda 1.η.3 S pomocí konstrukce `if ... then ... else ...` definujte funkci `firstNonZero :: Integer -> Integer -> Integer`, která vezme dvě celá čísla; pokud je první z nich nenulové, tak je vrátí, v opačném případě vrátí to druhé.

```
firstNonZero 1 4 ⇝* 1          firstNonZero 0 0 ⇝* 0
firstNonZero 0 4 ⇝* 4
```

*
* *

> **Pan Fešák doporučuje:** Priorita operátorů je jednou z věcí, které jsou nutné pro správné pochopení vyhodnocování výrazů. Je vhodné naučit se používat dotaz `:i`, který mimo jiné obsahuje i prioritu zadaného operátoru.
>
> Příklad dotazu v interpretu (řádek začínající `>` je zadán uživatelem):
>
> ```
> > :i *
> class Num a where
>   ...
>   (*) :: a -> a -> a
>   ...
>         -- Defined in ‘GHC.Num’
> infixl 7 *
> ```
>
> Pro nás je nyní podstatný poslední řádek, který říká, že se jedná o infixový operátor, a popisuje jeho prioritu a asociativitu. V tomto případě `infixl 7 *` znamená, že se jedná o operátor priority 7, který se závorkuje (asociuje) zleva (`infixl`; zprava by bylo `infixr`), tedy například `2 * 3 * 7` se vyhodnocuje jako by bylo uzávorkované `(2 * 3) * 7`. Například pro `==` dostaneme `infix 4 ==`, což znamená, že `==` nelze řetězit s dalšími operátory na stejné úrovni priority a jeho priorita je 4.
>
> Pokud nemá operátor (funkce) explicitně definovanou prioritu, jeho priorita je 9 a je asociativní zleva. I některé binární funkce zapsané písmeny mají určenou prioritu a asociativitu, kterou využijí, pokud jsou zapsány infixově. Příkladem takové funkce je `div`.
>
> Konečně prefixová aplikace má vždy přednost před aplikací infixovou, například ve výrazu `div 3 2 ^ 4` se provede nejprve dělení a až pak umocňování, jako by byl výraz uzávorkovaný `(div 3 2) ^ 4`.

Etuda 1.η.4 S použitím interpretu jazyka Haskell porovnejte vyhodnocení následujících dvojic výrazů a rozdíl vysvětlete.

a) `5 + 9 * 3` versus `(5 + 9) * 3`
b) `2 ^ 2 ^ 2 == (2 ^ 2) ^ 2` versus `3 ^ 3 ^ 3 == (3 ^ 3) ^ 3`
c) `3 + 3 + 3` versus `3 == 3 == 3`
d) `("Haskell" == "je") == "super"` versus `('a' == 'a') == ('a' == 'a')`

> **Pan Fešák doporučuje:** Operátory a funkce se můžou vyskytovat v prefixové i infixové verzi. Proto si dohledejte, jaký je rozdíl mezi `*` vs. `(*)` a `mod` vs. `` `mod` ``.

Etuda 1.η.5 Definujte s využitím více definičních rovností funkci `isWeekendDay :: String -> Bool`, která rozhodne, jestli je daný řetězec jméno víkendového dne.

```
isWeekendDay "Saturday" ⇝* True
isWeekendDay "Monday" ⇝* False
isWeekendDay "apple" ⇝* False
```

Etuda 1.η.6 Určete typy následujících výrazů a najděte další výrazy stejného typu. Své řešení si ověřte s pomocí interpretu.

a) `'a'`
b) `"Don't Panic."`
c) `not`
d) `(&&)`
e) `True`
f) `('a', True)`

1.1 Jednoduché funkce, `if`, `where`/`let ... in ...`

> **Pan Fešák doporučuje:** Pokud jste si ještě nevytvořili soubor pro toto cvičení, teď je dobrý čas jej vytvořit. Funkce z dalších příkladů pište do souboru a pak je testujte v GHCi.

Př. 1.1.1 S využitím interního příkazu `:info` interpretu GHCi zjistěte prioritu a asociativitu následujících operací:

`^`, `*`, `/`, `+`, `-`, `==`, `/=`, `>`, `<`, `>=`, `<=`, `&&`, `||`

Př. 1.1.2 Vytvořte funkci `circleArea :: Double -> Double`, která pro zadaný poloměr spočítá obsah kruhu o tomto poloměru. Přibližná hodnota konstanty π se dá v Haskellu získat pomocí konstanty `pi`.

Př. 1.1.3 Napište funkci `snowmanVolume :: Double -> Double -> Double -> Double`, která spočítá celkový objem sněhuláka s koulemi o zadaných poloměrech. Vzpomeňte si, že objem koule s poloměrem r je $\frac{4}{3} \cdot \pi \cdot r^3$. Přibližná hodnota konstanty π se dá v Haskellu získat pomocí konstanty `pi`. V řešení zkuste vhodně využít lokální definici (`where` nebo `let ... in ...`).

```
snowmanVolume 1 1 1      ⇝* 12.566370614359172
snowmanVolume 1 2 3      ⇝* 150.79644737231007
snowmanVolume 1 0 0      ⇝* 4.1887902047863905
```

> **Jindřiška varuje:** V Haskellu odsazujeme vždy mezerami, nikoli tabulátory. Používání tabulátorů rychle vede ke kombinaci obou variant a velmi podivným chybovým hláškám.

> **Jindřiška varuje:** V porovnání s jinými (převážně imperativními) jazyky má Haskell striktnější pravidla pro konstrukci `if ... then ... else ...` a je nutné je všechna znát.
>
> V imperativních jazycích totiž Haskellovému výrazu `if p then t else f` neodpovídá řídicí příkaz `if p then t else f`, ale ternární operátor: `p ? t : f` (nejen) v jazyce C nebo `t if p else f` v Pythonu.

Př. 1.1.4 1.1.4 Pro zadané tři kladné délky stran trojúhelníka rozhodněte, zda se jedná o pravoúhlý trojúhelník. Pravoúhlý trojúhelník je možné poznat tak, že pro délky jeho stran platí Pythagorova věta (tedy součet druhých mocnin dvou kratších stran je roven druhé mocnině nejdelší strany). V řešení zkuste vhodně využít lokální definici (`where` nebo `let ... in ...`).

```
isRightTriangle 3 4 5     ⇝* True      isRightTriangle 70 42 56 ⇝* True
isRightTriangle 42 42 42  ⇝* False     isRightTriangle 25 24 7  ⇝* True
```

Př. 1.1.5 Definujte funkci `max3 :: Integer -> Integer -> Integer -> Integer`, která pro tři celá čísla `x`, `y` a `z` vrátí to největší z nich. Naprogramujte dvě verze, jednu pomocí funkce `max` a jednu pomocí `if ... then ... else ...`.

Př. 1.1.6 Naprogramujte funkci `mid :: Integer -> Integer -> Integer -> Integer`, která pro tři celá čísla `x`, `y` a `z` vrátí to *prostřední* z nich (tj. to druhé v jejich uspořádané trojici podle $\leq$).

```
mid 1 2 3 ⇝* 2                  mid 15 113 111 ⇝* 111
mid 42 16 69 ⇝* 42              mid 42 42 42 ⇝* 42
```

Př. 1.1.7 Pomocí `if` a funkce `mod` definujte funkci `tell :: Integer -> String`, která bere jako argument jedno kladné celé číslo `n` a vrací:

- `"one"` pro `n = 1`,
- `"two"` pro `n = 2`,
- `"(even)"` pro sudé `n > 2` a
- `"(odd)"` pro liché `n > 2`

Př. 1.1.8 U následujících výrazů rozhodněte, zda jsou správně, a pokud jsou špatně, zdůvodněte proč a vhodným způsobem je upravte.

a) `if 5 - 4 then False else True`
b) `if 0 < 3 && odd 6 then 0 else "FAIL"`
c) `(if even 8 then (&&)) (0 > 7) True`
d) `if 42 < 42 then (&&) else (||)`

> **Pan Fešák doporučuje:** Pokud v konstrukci `if ... then ... else ...` mají větve `then` a `else` typ `Bool`, pak je vhodné se zamyslet nad tím, zda celá konstrukce není zbytečná a problém nelze vyřešit použitím vhodných logických operátorů.
>
> Příkladem je výraz `if` *`podmínka`* `then True else False`, který se dá zjednodušit na výraz *`podmínka`*.

1.2 Priority operátorů a volání funkcí

Př. 1.2.1 Doplňte všechny implicitní závorky do následujících výrazů:

a) `recip 2 * 5`
b) `sin pi + 2 / 5`
c) `f g 3 * g 5 `mod` 7`
d) `42 < 69 || 5 == 6`
e) `2 + div m 18 == m ^ 2 ^ n && m * n < 20`

> **Pan Fešák doporučuje:** Pokud nevíte, co daná funkce nebo operátor dělá, je vhodné si ji najít v dokumentaci. Příkladem může být operátor `(||)`.

Př. 1.2.2 Přepište infixové zápisy výrazů do syntakticky správných prefixově zapsaných výrazů a naopak:

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

Př. 1.2.3 Doplňte všechny implicitní závorky do následujících výrazů:

1.2.3

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

Př. 1.2.4 Zjistěte (bez použití interpretu), na co se vyhodnotí následující výraz. Poté všech sedm funkcí přepište do prefixového tvaru a pomocí interpretu ověřte, že se hodnota výrazu nezměnila.

```
5 + 7 * 5 `mod` 3 `div` 2 == 3 * 2 - 1
```

Př. 1.2.5 Do následujícího výrazu doplňte implicitní závorky a pak převeďte všechny operátory v něm do prefixového tvaru.

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

1.3 Definice s více definičními rovnostmi

Také nazývána **definice podle vzoru** podle toho, že smysluplná definice s více definičními rovnostmi musí využívat různé vzory v různých definičních rovnostech.

Př. 1.3.1 Bez použití podmíněného výrazu `if ... then ... else ...` definujte funkce

```
logicalNot :: Bool -> Bool
logicalAnd :: Bool -> Bool -> Bool
logicalOr  :: Bool -> Bool -> Bool
```

které se chovají stejně jako funkce logické negace, konjunkce a disjunkce.

Nesmíte využít žádné logické funkce definované v Haskellu.

Př. 1.3.2 Definujte funkci `isSmallVowel :: Char -> Bool`, která rozhodne, jestli je dané písmeno malou samohláskou anglické abecedy. Znakové literály se v Haskellu píší do apostrofů, například literál znaku *a* se v Haskellu zapíše jako `'a'`.

> **Jindřiška varuje:** V Haskellu, na rozdíl od například Pythonu (a mnoha dalších skriptovacích jazyků), je rozdíl mezi věcmi uzavřenými mezi apostrofy `'...'` a uvozovkami `"..."`. Mezi apostrofy je uzavřen vždy jeden znak (typu `Char`), zatímco mezi uvozovkami se jedná o řetězec (typ `String`; řetězec se skládá ze znaků).

Př. 1.3.3 Naprogramujte funkci `solveQuad :: Double -> Double -> Double -> (Double, Double)`, která vezme na vstupu koeficienty `a`, `b`, `c` a vyřeší kvadratickou rovnici $ax^2 + bx + c = 0$ s v reálných číslech. Připomeňme, jak se řeší kvadratická rovnice:

- Pokud $a \neq 0$, je třeba spočítat diskriminant $d = b^2 - 4ac$. Řešení pak jsou $\frac{-b-\sqrt{d}}{2a}$ a $\frac{-b+\sqrt{d}}{2a}$. Tato řešení vraťte ve dvojici (může se stát, že řešení budou stejná, vždy však musíme vrátit dvojici).
- Pokud $a = 0$, pak jde o lineární rovnici a výsledek je $x = -\frac{c}{b}$. Vraťte dvě identické hodnoty ve dvojici.
- Situaci, kdy diskriminant vychází záporný či $a = b = 0$, neřešte (vůbec tento případ

nezohledňujte).

V řešení vhodně využijte více definičních rovností a lokální definice.

```
solveQuad 1 0 (-1) ⇝* (-1.0, 1.0)
solveQuad 0 2 2 ⇝* (-1.0, -1.0)
solveQuad 1 (-6) 5 ⇝* (1.0, 5.0)
```

Př. 1.3.4 
Naprogramujte funkci `parallelToAxis`, která o úsečce zadané souřadnicemi bodů v rovině rozhodne, jestli je rovnoběžná s jednou ze souřadnicových os roviny. Můžete předpokládat, že vstup skutečně představuje úsečku, tedy že vstupní body jsou různé.

Poznámka: Vyhněte se funkcím `fst` a `snd` a použijte výhradně vzory.

```
parallelToAxis :: (Integer, Integer) -> (Integer, Integer) -> Bool
parallelToAxis (0, 0) (1, 1) ⇝* False
parallelToAxis (1, 1) (1, 4) ⇝* True
parallelToAxis (16, 42) (12, 19) ⇝* False
parallelToAxis (4, 2) (9, 2) ⇝* True
```

Př. 1.3.5 
Vzpomeňte si na příklad 1.1.7 a zkuste jej napsat pomocí více definičních rovností.

1.4 Základní typy

> **Pan Fešák doporučuje:** Příkaz GHCi `:t` je dobrý pro kontrolu, ale je vždy lepší být si schopen určit typy sám. A stejně tak je dobré u funkcí typy vždy psát. Tím si procvičujete přemýšlení v typech a budete moci plně využít toho, že vám je Haskell umí kontrolovat, a může tak poznat rozdíl mezi vaší představou o tom, co má funkce dělat (reflektovanou v typu, který jste napsali), a co skutečně dělá (což je reflektované v typu, který si GHC odvodí).

Př. 1.4.1 

Nalezněte příklady hodnot následujících typů:

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

Př. 1.4.2 

Určete typy následujících výrazů.

a) `True`
b) `"True"`
c) `not True`
d) `True || False`
e) `True && ""`
f) `fun 1`, kde funkce `fun` je nějaká funkce typu

```
fun :: Integer -> Integer
```

g) `fun 3.14`, kde `fun` je stejného typu jako v části f)
h) `solve 3 8`, kde `solve` je nějaká funkce typu

```
solve :: Int -> Int -> Int
```

Př. 1.4.3 Určete typ následujících funkcí, které jsou definovány předpisem:

1.4.3

a)
```
implication a b = not a || b
```

b)
```
foo _   "42" = True
foo 'a' _    = True
foo _   _    = False
```

c)
```
ft True x y = False
ft x    y z = y
```

Př. 1.4.4 Naprogramujte funkci `isDivisibleBy :: Integral a => a -> a -> Bool`, která pro dvě celá čísla `x` a `y` rozhodne, zda je `x` dělitelné `y`.

> `Integral a =>` je omezení na typy, které lze dosadit za typovou proměnnou `a`. V tomto případě za `a` lze dosadit celočíselné typy, například `Int`, `Integer`. Obdobně například omezení `Num a =>` znamená, že za typovou proměnnou `a` lze dosadit číselný typ (třeba `Integer` nebo `Double`).

> **Jindřiška varuje:** U operací *dělení* (a *zbytek po dělení*) je nutné rozlišovat operace pracující s celými čísly (`div` a `mod`) a dělení desetinných čísel (`/`).

Př. 1.4.5 Vysvětlete, jak je možné, že výrazy `(3 / 2)` a `(15 / 3)` vyprodukují desetinné číslo i přes to, že jejich argumenty jsou celá čísla, zatímco výraz `(3 + 2)` vyprodukuje celé číslo. *Poznámka*: Může se vám hodit podívat se na typy různých aritmetických operací.

*
* *

> **Na konci cvičení byste měli zvládnout:**
>
> - ovládat základní příkazy interpretu GHCi a prakticky je využívat;
> - otypovat jednoduché výrazy a funkce;
> - vytvářet jednoduché funkce pracující s čísly a pravdivostními hodnotami;
> - vytvářet funkce s více definičními rovnostmi;
> - prakticky umět využít podmíněný výraz `if` a lokální definice pomocí `let ... in ...` nebo `where`.

Řešení

Řeš. 1.η.2 Potřebujeme zjistit, jestli se `y` rovná číslu o jedna většímu než `x`.

```
isSucc :: Integer -> Integer -> Bool
isSucc x y = y == x + 1
```

Řeš. 1.η.4 a) V důsledku priorit operátorů je implicitní závorkování kolem násobení, tj. `5 + (9 * 3)`.

b) Operace umocňování asociuje zprava, tedy v případě více výskytů (`^`) za sebou se implicitně závorkuje zprava. Obecně tedy

`n ^ n ^ n`	=	`n ^ (n ^ n)`	≠	`(n ^ n)^ n`	=	`n ^ (n ^ 2)`
n^{n^n}	=	$n^{(n^n)}$	≠	$(n^n)^n$	=	$n^{(n^2)}$

c) Tady se setkáváme s případem, kdy je operátor neasociativní, tedy není definováno, jak se výraz zpracovává v případě výskytu více operátorů stejné priority vedle sebe, a výraz je tedy nekorektní. Důvod neasociativity je jednoduchý: u (`==`) totiž nemá smysl definovat asociativitu, protože jeho výsledek je typu `Bool`, ale argumenty mohou být jiného typu – to nakonec vidíme i na našem příkladě `3 == 3 == 3`, ať jej uzávorkujeme libovolně, nebudou nám sedět typy (budeme porovnávat číslo a `Bool`).

d) V případě, že explicitně uvedeme závorkování pro relační operátory, dostaneme se do obdobné situace jako v předchozím podpříkladu, tedy porovnání logické hodnoty a řetězce, což v Haskellu nelze. Naproti tomu výsledkem porovnání `'a' == 'a'` dostaneme v obou případech logickou hodnotu, a ty mezi sebou porovnávat můžeme, protože jsou stejného typu. Všimněte si, že v tomto výrazu se vyskytuje == jednou ve verzi pro znaky (`'a' == 'a'`) a jednou pro `Bool` (prostřední výskyt).

Řeš. 1.η.5 Je potřeba se zamyslet nad tím, které případy má smysl vytáhnout pomocí vzorů jako ty „speciální“. V našem případě to budou právě víkendové dny a pro vše ostatní vrátíme `False`.

```
isWeekendDay "Saturday" = True
isWeekendDay "Sunday"   = True
isWeekendDay _          = False
```

Řeš. 1.η.6 Typ snadno ověříte pomocí příkazu `:t` k otypování výrazu v GHCi (typ `[Char]` je ekvivalentní typu `String`).

a) `'a' :: Char`; libovolný Unicode znak je stejného typu: `'ľ'`, `'ř'` nebo speciální znak nového řádku `'\n'`.

b) `"Don't Panic." :: String` (nebo ekvivalentně `[Char]`); `""`, `"a"` nebo `"nejvnějšnější"`.

c) `not :: Bool -> Bool`; např. funkce

```
boolId :: Bool -> Bool
boolId x = x
```

d) `(&&) :: Bool -> Bool -> Bool`; např. `(||)`

e) `True :: Bool`; např. `42 == 6 * 7`

f) `(Char, Bool)`; např. `('b', False)`

Řeš. 1.1.1 Celkově zjistíme, že operátory jsou uvedeny v zadání v pořadí od nejvyšší priority (9) až k nejnižší (1).

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

Řeš. 1.1.2 Obsah kruhu o poloměru r se vypočítá vzorečkem πr^2. Do Haskellu to snadno přepíšeme jako:

```
circleArea :: Double -> Double
circleArea r = pi * r ^ 2
```

Řeš. 1.1.3 Použijeme lokální definici `sphereVolume :: Double -> Double` pro výpočet objemu jedné koule.

```
snowmanVolume :: Double -> Double -> Double -> Double
snowmanVolume a b c = sphereVolume a + sphereVolume b + sphereVolume c
    where
        sphereVolume r = 4 * pi * (r ^ 3) / 3
```

Řeš. 1.1.4 Nejprve zjistíme, která strana je nejdelší, a pak strany pošleme ve správném pořadí do rovnice pro Pythagorovu větu, kterou si zadefinujeme pomocí lokální definice.

```
isRightTriangle :: Integer -> Integer -> Integer -> Bool
isRightTriangle x y z = if x >= y && x >= z
    then pyt y z x      -- x is the maximum
    else if y >= z      -- x is not the maximum, one of y or z must be
         then pyt x z y -- y is the maximum
         else pyt x y z -- z is the maximum
    where
        pyt a b c = a ^ 2 + b ^ 2 == c ^ 2
```

Alternativou je vyzkoušet všechny tři možné volby pro nejdelší stranu (víme jistě, že pokud vybereme některou z kratších stran místo té nejdelší, rovnost v Pythagorově větě nevyjde). I zde můžeme s výhodou využít lokální definice.

```
isRightTriangle' :: Integer -> Integer -> Integer -> Bool
isRightTriangle' x y z = pyt x y z || pyt y z x || pyt x z y
    where
        pyt a b c = a ^ 2 + b ^ 2 == c ^ 2
```

Řeš. 1.1.5

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z = max x (max y z)

max3' :: Integer -> Integer -> Integer -> Integer
max3' x y z = if x > y
    then if z > x then z else x
    else if z > y then z else y
```

Alternativní řešení pomocí konstrukce `if ... then ... else ...`:

```
max3'' :: Integer -> Integer -> Integer -> Integer
max3'' x y z = if x > y && x > z
    then x
    else if z > y then z else y
```

Řeš. 1.1.6 Přímočaré je řešení pomocí `if` a `min`/`max`, stačí si uvědomit, že prostřední číslo je menší nebo rovno maximu a větší nebo rovno minimu:

```
mid :: Integer -> Integer -> Integer -> Integer
mid x y z = if min y z <= x && x <= max y z
    then x
    else if min x z <= y && y <= max x z
        then y
        else z
```

Alternativou je vypočítat prostřední číslo za pomoci součtu, minima a maxima:

```
mid' :: Integer -> Integer -> Integer -> Integer
mid' x y z = x + y + z - max z (max x y) - min z (min x y)
```

Další alternativou je využít sílu funkce `max3`, kterou jsme již definovali (předpokládejme tedy, že je definována). Ta nám umožňuje vybírat to největší z libovolných tří celých čísel. Pokud tedy vybereme čísla z původní trojice tak, že máme zaručeno, že vybereme všechna kromě toho největšího (některé se může zopakovat), maximem z těchto čísel bude právě prostřední prvek v uspořádání podle $\leq$:

```
mid'' :: Integer -> Integer -> Integer -> Integer
mid'' x y z = max3 (min x y) (min y z) (min x z)
```

Řeš. 1.1.7 Řešení je zdlouhavé, ale celkem přímočaré. Pokud víme, že číslo n je sudé, právě když $n \bmod 2 = 0$, stačí jenom zbylé podmínky vypsat do zanořených `if`ů a dát pozor na to, že `if` v Haskellu má vždy i neúspěšnou větev po `else`. Zároveň obě větve musí být stejného *typu* (co intuitivně znamená, že obě musí vracet číslo nebo znak nebo řetězec…):

```
tell :: Integer -> String
tell n = if n > 2
         then if mod n 2 == 0 then "(even)" else "(odd)"
         else if n == 1 then "one" else "two"
```

Vnořené `if`y lze i uzávorkovat, ale je to zbytečné.

Toto řešení je poněkud špatně čitelné a použití `if` v Haskellu není vždy žádoucí. Časem si ukážeme něco lepšího.

Řeš. 1.1.8

a) Podmínka musí být logický výraz typu `Bool`, což výraz `5 - 4` není – jde o výraz celočíselného typu. Haskell nikdy sám nekonvertuje výrazy jednoho typu na druhý. Vhodná úprava celého výrazu je pak třeba `5 - 4 == 0`.

b) Výrazy v `then` a `else` větvi musí být stejného typu, protože celý podmínkový výraz musí mít vždy stejný typ bez ohledu na hodnotu podmínky. Výraz lze opravit na

```
if 0 < 3 && odd 6 then "OK" else "FAIL"
```

což už je typově správně.

c) Na první pohled podivně vypadající konstrukce, kde výsledkem podmínkového výrazu je operátor `(&&)`, je správná. V Haskellu jsou funkce/operátory rovnocenné s číselnými či jinými konstantami. Problémem je chybějící větev `else`. Podmíněný výraz má syntaktické omezení, že vždy musí obsahovat jak `then`, tak `else` větev, i když by podmínka zaručovala použití jen jedné z nich. Kdyby podmínka mohla být vyhodnocena na nepravdu a chybělo by `else`, pak by výraz neměl žádnou hodnotu, kterou by vrátil. Ale výraz v Haskellu vždy musí mít nějakou hodnotu a je tedy třeba přidat `else` větev:

```
if even 8 then (&&) else (||)
```

d) Tento výraz je v pořádku. A to i přes to, že v interpretu dostaneme podivnou hlášku:

```
> if 42 < 42 then (&&) else (||)

<interactive>:1:1: error:
    • No instance for (Show (Bool -> Bool -> Bool))
        arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
```

Tato hláška však jen říká, že výslednou hodnotu výrazu nelze vypsat (kritická je tu ta část „`In a stmt of an interactive GHCi command: print it`“, která říká, že se jedná o chybu při vypisování výsledku výrazu). Konkrétně zde se interpret snaží vypsat hodnotu typu `Bool -> Bool -> Bool`, tedy binární funkci, která bere dvě pravdivostní hodnoty a jednu vrací. Funkce ale nelze v GHCi za normálních okolností vypisovat. Pokud nebudete chtít výsledek výrazu vypsat, ale jen ho otypujete pomocí příkazu `:t if 42 < 42 then (&&) else (||)`, k žádné chybě nedojde.

Řeš. 1.2.1 Při doplňování implicitních závorek je potřeba se řídit prioritou/asociativitou infixově zapsaných operátorů a závorkováním aplikace funkcí na argumenty. Postupujeme následovně:

1. Obsahuje-li výraz infixově zapsané operátory, najdeme ty s nejnižší prioritou, které nejsou v závorkách, a jejich operandy uzávorkujeme. Pokud je těchto operátorů více než jeden, jednotlivé operandy závorkujeme dle asociativity daných operátorů (pokud se vedle sebe vyskytují dva operátory se stejnou prioritou, ale různou asociativitou nebo bez asociativity, výraz je nesprávně utvořený).
2. Pokud již ve výrazu nejsou infixové operátory, tj. výraz je jednoduchý (konstanta, proměnná nebo název funkce), prefixová aplikace funkce nebo aplikace infixově zapsaného binárního operátoru na dva jednoduché argumenty, skončili jsme.
3. V opačném případě aplikujeme stejný postup na všechny podvýrazy vzniklé buď doplněnými závorkami, nebo dosud nezpracovanými závorkami v původním výrazu.

Řešení jednotlivých příkladů budou následující:

a) `(recip 2) * 5`
b) `(sin pi) + (2 / 5)`
c) `((f g 3) * (g 5)) `mod` 7` (funkce `f` je aplikována na 2 argumenty; asociativita)
d) `(42 < 69) || (5 == 6)`
e) `((2 + (div m 18)) == (m ^ (2 ^ n))) && ((m * n) < 20)`

Řeš. 1.2.2 Je důležité zachovávat pořadí operandů! I když jde o komutativní operátor, nelze obecně při změně mezi prefixovým a infixovým zápisem měnit jejich pořadí, protože vzniklý výraz nebude totožný. Navíc Haskell nijak negarantuje, že např. + je komutativní.

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

Řeš. 1.2.3

a) `f . (g x)`
b) `2 ^ (mod 9 5)`
c) `f . (((.) g h) . id)`
d) `((2 + (((div m 18) * m) `mod` 7)) == (((m ^ (2 ^ n)) - m) + 11))`
`&& ((m * n) < 20)`

e) `(f 1 2 g) + (((+) 3) `const` (g f 10))`
f) `(replicate 8 x) ++ ((filter even) (enumFromTo 1 (3 + (9 `mod` x))))`
g) `(id id) . (flip const const)`

Řeš. 1.2.4 Nejdříve podle priority operátorů do výrazu zapíšeme implicitní závorky (kvůli různým prioritám operátorů):

```
(5 + (((7 * 5) `mod` 3) `div` 2)) == ((3 * 2) - 1)
```

Pak už lehce zjistíme, že výraz se vyhodnotí na `False`.

Při vyhodnocování výrazu v zadání se jako poslední vyhodnotí funkce s nejnižší prioritou, v našem případě `(==)`. Přepíšeme tedy do prefixu nejdříve tuto funkci:

```
(==) (5 + 7 * 5 `mod` 3 `div` 2) (3 * 2 - 1)
```

Následně v každém z argumentů opět najdeme funkci s nejnižší prioritou – v prvním je to funkce `(+)`, ve druhém pak `(-)`. Přepisem těchto funkcí do prefixu dostaneme:

```
(==) ((+) 5 (7 * 5 `mod` 3 `div` 2)) ((-) (3 * 2) 1)
```

Stejným způsobem pokračujeme i nadále. Jestliže narazíme na skupinu operátorů se stejnou prioritou (například `(*), mod, div`), ověříme si jejich směr sdružování (závorkování). V našem případě se sdružuje (závorkuje) zleva. To v praxi znamená, že jako poslední se vyhodnotí funkce `div`. Výraz tedy přepíšeme následovně:

```
div (7 * 5 `mod` 3) 2
```

Stejným způsobem pokračujeme, dokud nám nezůstanou žádné infixově zapsané operátory:

```
(==) ((+) 5 (div (mod ((*) 7 5) 3) 2)) ((-) ((*) 3 2) 1)
```

Řeš. 1.2.5 Se závorkami:

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

A po přepisu do prefixu:

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

Řeš. 1.3.1 a)

```
logicalNot :: Bool -> Bool
logicalNot True  = False
logicalNot False = True
```

b)

```
logicalAnd :: Bool -> Bool -> Bool
logicalAnd True True = True
logicalAnd _    _    = False
```

c)

```
logicalOr :: Bool -> Bool -> Bool
logicalOr False False = False
logicalOr _     _     = True
```

Řeš. 1.3.2 Opět se zamyslíme nad tím, které případy vytáhnout do vzorů. V tomto případě to budou samohlásky, protože samohlásek je v anglické abecedě oproti souhláskám značně méně. Pro všechno ostatní jenom jednoduše vrátíme `False`:

```
isSmallVowel 'a' = True
isSmallVowel 'e' = True
isSmallVowel 'i' = True
```

```
isSmallVowel 'o' = True
isSmallVowel 'u' = True
isSmallVowel _   = False
```

Řeš. 1.3.3 Případ $a = 0$ je vhodné oddělit do samostatné definiční rovnosti. Pro diskriminant a pro řešení lineárního případu je vhodné použít lokální definici.

```
solveQuad :: Double -> Double -> Double -> (Double, Double)
solveQuad 0 b c = let x = -c / b in (x, x)
solveQuad a b c = let sd = sqrt (b * b - 4 * a * c)
                  in ((- b - sd) / (2 * a), (- b + sd) / (2 * a))
```


Všimněte si, že v 2. řádku definice ukládáme do lokální proměnné odmocninu z diskriminantu a nikoli diskriminant jako takový. To proto, abychom nemuseli odmocninu počítat dvakrát. Rovněž si můžete všimnout, že složky dvojice v témže řádku jsou velmi podobné. To můžeme napravit například tak, že si definujeme pomocnou funkci:

```
solveQuad' :: Double -> Double -> Double -> (Double, Double)
solveQuad' 0 b c = let x = -c / b in (x, x)
solveQuad' a b c = let sd = sqrt (b * b - 4 * a * c)
                       res y = (- b + y) / (2 * a)
                   in (res (- sd), res sd)
```

Řeš. 1.3.4 Spojnice bodů bude rovnoběžná s osou, jestliže buď $x$ová souřadnice obou bodů bude stejná (pak bude spojnice rovnoběžná s osou x), nebo $y$ová souřadnice bude stejná.

```
parallelToAxis :: (Integer, Integer) -> (Integer, Integer) -> Bool
parallelToAxis (x1, y1) (x2, y2) = x1 == x2 || y1 == y2
```

Řeš. 1.3.5 Řešení pomocí více definičních rovností bývá obvykle více čitelné, zde se zbavíme zanořených podmínek.

```
tell' :: Integer -> String
tell' 1 = "one"
tell' 2 = "two"
tell' n = if mod n 2 == 0 then "(even)" else "(odd)"
```

Řeš. 1.4.1

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 interpretu 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) `(1, 1)`, `(42, 16)`, `(10 - 5, 10 ^ 10000)`, …
Libovolná dvojice celých čísel. Pokud jako `Int` zvolíte dostatečně velké číslo pak vám může „přetéct“. Toto rozmezí můžete otestovat zadáním

```
> (minBound, maxBound) :: (Int, Int)
```

do svého interpretu. `Integer` je omezen pouze pamětí počítače.

f) `(0, 3.14, True)`, …
Trojice, složky musí odpovídat typům.

g) `()`
Takzvaná nultice je typem s jedinou hodnotou. Typ `()` někdy také označujeme jako jednotkový typ nebo v angličtině *unit*. 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.

h) `((), (), ())`
Jediná možná hodnota je trojice, jejímž každým prvkem je nultice.

Řeš. 1.4.2 a) `Bool`, výraz je hodnotovým 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: `True :: Bool`, `"" :: String`, `(&&) :: Bool -> Bool -> Bool`. 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 `fun`.

g) Nesprávně utvořený výraz. Výraz `3.14` nemůže být typu `Integer`, protože se nejedná o celé číslo, tedy jej nelze dosadit do funkce `fun`.

h) `Int`, protože funkce bere dva parametry typu `Int` a `Int` vrací. Výrazy `3` i `8` mohou být `Int`, a tedy je lze dosadit jako parametry.

Řeš. 1.4.3 a) Výraz `not a || b` se uzávorkuje `(not a) || b`, přičemž operátor `(||)` má typ `Bool -> Bool -> Bool`, tedy výraz `not a` má být typu `Bool` a i výraz `b` je typu `Bool`. Nyní stačí jenom určit typ výrazu `a`, který známe z toho, že výraz `not` je typu `Bool -> Bool`, tedy výraz `a` je typu `Bool`. Platí tedy `implication :: Bool -> Bool -> Bool`.

b) Funkce `foo` bere dva argumenty a jejím výsledkem je výraz typu `Bool`. Z prvního řádku předpisu víme, že druhý argument je typu `String`, a ze druhého řádku předpisu víme, že první argument je typu `Char`. Funkce `foo` má tedy typ `Char -> String -> Bool`.

c) Z prvního řádku definice vidíme, že první argument musí být typu `Bool` a funkce taktéž vrací `Bool`. Z druhého řádku pak vidíme, že typ druhého argumentu musí být stejný jako typ návratové hodnoty. O typu třetího argumentu nevíme vůbec nic – může to

být cokoli. Celkově tedy dostáváme `ft :: Bool -> Bool -> a -> Bool`.

Řeš. 1.4.5 Důležitý je zde typ operátoru `(/) :: Fractional a => a -> a -> a`. Pro srovnání uveďme další příklady aritmetických operací: `(+) :: Num a => a -> a -> a`, `div :: Integral a => a -> a -> a`. Všimněte si, že zatímco u sčítání máme typové omezení, že argumenty jsou čísla, u dělení pomocí `(/)` jsou to desetinná čísla[1]. U celočíselného dělení pak můžeme používat jen celá čísla. Právě tato omezení způsobí, že ve výrazech jako `(3 / 2)` se odvodí, že argumenty dělení musí být typy schopné reprezentovat desetinná čísla.

Ve skutečnosti celočíselný *literál* (tedy zápis čísla v kódu) může být libovolného číselného typu podle potřeby kódu, v němž je použit. Desetinný literál (např. `3.14`) pak může být libovolného desetinného typu. Odvozování typů z operací funguje samozřejmě i při zanořování operací – u výrazu `((7 + 8) / 5)` bude výsledkem desetinné číslo a i sčítání se bude provádět nad desetinnými čísly.

Poznámka: Na to, aby se operace mohla provést musí nakonec interpret odvodit konkrétní typ, v němž se výraz vyhodnotí, nestačí mu tedy jen vědět, že se jedná například o nějaký desetinný typ. Zde platí jednoduché pravidlo, celá čísla se bez dalších omezení vyhodnocují v typu `Integer` a desetinná v typu `Double`. *Tento postup však nijak neovlivňuje typ výrazu a přichází na řadu až při vyhodnocování v interpretu.*

[1] Přesněji jde o typy, pomocí niž lze reprezentovat zlomky. Tyto typy jsou definovány právě tím, že umožňují dělení.