

Cvičení 4: Vlastní a rekurzivní datové typy, Maybe

Před čtvrtým cvičením je zapotřebí:

- ▶ znát koncept datových typů:
 - ▷ *hodnotový* a *typový* konstruktor;
 - ▷ klíčová slova `data` a `type` a rozdíl mezi nimi;
 - ▷ definice funkcí pomocí vzorů pro vlastní datové typy;
- ▶ znát datový typ `Maybe`;
- ▶ mít základní znalosti o *stromech* – pojmy *kořen*, *cesta*, *hloubka vrcholu*.

Etudy

Etuda 4.η.1 Mějme datový typ `Day` představující dny v týdnu definovaný níže. Definujte funkci `weekend :: Day -> Bool`, která o zadaném dni určí, jestli je to víkendový den. Datový typ `Day` je definován takto:

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

Etuda 4.η.2 Mějme následující definici:

```
data Object = Cube Double Double Double -- a, b, c
  | Cylinder Double Double -- r, v
```

- a) Uvedte příklady hodnot, které mají typ `Object`.
- b) Kolik je v definici použito hodnotových konstruktorů a které to jsou?
- c) Kolik je v definici použito typových konstruktorů a které to jsou?
- d) Definujte funkce `volume` a `surface`, které pro hodnoty uvedeného typu počítají objem, respektive povrch.

Příklady vyhodnocení korektně definovaných funkcí jsou:

```
volume (Cube 1 2 3) ~>* 6.0
surface (Cylinder 1 3) ~>* 25.132741228718345
```

V řešení můžete použít konstantu `pi`.

Etuda 4.η.3 Uvažte následující kód:

```
data Contained a = NoValue | Single a | Pair a a
data Compare a = SameContainer
  | SameValue (Contained a)
  | DifferentContainer

cmpContained :: Eq a => (Contained a)
  -> (Maybe (Contained a))
  -> (Compare a)
```


```

cmpContained (NoValue) (Just (NoValue)) =
  (SameValue (NoValue))
cmpContained (Single x1) (Just (Single x2)) =
  if (x1 == x2)
  then (SameValue (Single x1))
  else (SameContainer)
cmpContained (Pair x1 y1) (Just (Pair x2 y2)) =
  if (x1 == x2) && (y1 == y2)
  then (SameValue (Pair x1 y1))
  else (SameContainer)
cmpContained _ _ = DifferentContainer

```

Vášim úkolem je odstranit čtrnáct párů závorek tak, aby kód stále šel zkompileovat a zároveň si zachoval funkcionalitu.

U každého odstraňovaného páru si zkuste zdůvodnit, proč na daném místě nemusí být explicitně uvedený, a u každého páru, který necháváte, zdůvodnit, proč na daném místě musí zůstat.

Etuda 4.η.4  Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. Korektnost uvažujte pouze v kontextu standardně definovaných datových typů a funkcí (tj. těch z module Prelude), bez dalších hypotetických definic.

- `Maybe Char`
- `Maybe 42`
- `Just "Dance!"`
- `Just Integer`
- `Just Nothing`
- `[Nothing, Just 4]`
- `\b matters -> if b then Nothing else matters`


Etuda 4.η.5 Definujte funkci `safeDiv :: Integral a => a -> a -> Maybe a`, která celočíselně podělí dvě čísla a ošetří případy dělení nulou, tedy například:

```

safeDiv 6 3 ~>* Just 2
safeDiv 12 0 ~>* Nothing
safeDiv (-4) (-4) ~>* Just 1

```

Etuda 4.η.6 Naprogramujte funkci `valOrElse :: Maybe a -> a -> a`, která vezme `Maybe`-hodnotu a výchozí hodnotu a vrátí buď hodnotu obsaženou v `Maybe` pokud taková existuje, nebo výchozí hodnotu pokud je v `Maybe` hodnota `Nothing`.

Etuda 4.η.7  Podívejte se na video shrnující některé na první pohled nejasné chybové hlášky interpretru jazyka Haskell. Můžete k tomu použít odkaz pod číslem tohoto příkladu; vede do studijních materiálů předmětu v ISu.

Pan Fešák doporučuje: Vlastní datové typy definované v příkladech této kapitoly najdete připravené k použití v souboru [04_data.hs](#) v příloze sbírky nebo ve studijních materiálech v ISu.

4.1 Vlastní datové typy

Př. 4.1.1 Definujeme následující datové typy:

```
>>=
type WeaponsPoints = Integer
type HealthPoints = Integer

data Armor = Leather | Steel -- kožené nebo ocelové brnění
    deriving (Eq, Show)

data Warrior = Warrior HealthPoints Armor WeaponsPoints
    deriving (Eq, Show)
```

- Uvedte příklady hodnot, které mají typ `Armor` a `Warrior`.
- Kolik je v definici použito hodnotových konstruktorů a které to jsou?
- Kolik je v definici použito typových konstruktorů a které to jsou?
- Definujte funkci `attack :: Warrior -> Warrior -> Warrior`. Prvním parametrem této funkce je útočník, druhým je obránce. Funkce vrací stav obránce po útoku. Každý bod útoku útočníka způsobí snížení zdraví obránce o jedna (ne ale níž než na nulu). Pokud používá obránce ocelové brnění, dělí se útočná síla zbraně dvěma. Předpokládejte, že hodnoty `HealthPoints` a `WeaponsPoints` jsou nezáporné.

Příklady vyhodnocení korektně definované funkce jsou:

```
warrior1 = Warrior 30 Leather 30
warrior2 = Warrior 20 Steel 25

attack warrior1 warrior2 ~>* Warrior 5 Steel 25
attack warrior2 warrior1 ~>* Warrior 5 Leather 30
```

Př. 4.1.2 Vytvořte nový datový typ `Jar` představující sklenici ve spíži. Každá sklenice je v jednom z následujících stavů:

4.1.jar



- je prázdná (`EmptyJar`);
- je v ní ovocná marmeláda (`Jam`), pamatujeme si typ ovoce, ze kterého byla vyrobena (`String`);
- jsou v ní okurky (`Cucumbers`), o nich si nemusíme nic pamatovat, stejně se hned snědí;
- je v ní kompot (`Compote`), pamatujeme si rok výroby (`Int`).

Vaší úlohou je pak nadefinovat funkci `stale :: Jar -> Bool`, která určí, jestli je obsah dané sklenice již zkažený. Prázdné sklenice, okurky ani marmelády se nekaží (možná je to tím, že se příliš rychle snědí), kompoty se pokazí za 10 let od zavaření (zadefinujte si celočíselnou konstantu `today`, ve které budete mít aktuální rok).

Pan Fešák doporučuje: Pro úplné pochopení principů vlastních datových typů a rozdílů mezi hodnotovými a typovými konstruktory je doporučeno projít a rozumět následujícím příkladům.

Př. 4.1.3 Identifikujte nově vytvořené typové a hodnotové konstruktory a určete jejich aritu.



- `data X = Value Int`
- ```
data M = A | B | N M
data N = C | D | M N
```
- `data Ha = Hah Int Float [Hah]`

- d) `data FMN = T (Int, Int) (Int -> Int) [Int]`  
 e) `type Fat = Float -> Float -> Float`  
 f) `data E = E (E, E)`

Př. 4.1.4 Které deklarace datových typů jsou správné?



- a) `data N x = NVal (x -> x)`  
 b) `type Makro = a -> a`  
 c) `data M = N (x, x) | N Bool | O M`  
 d) `type Fun a = a -> (a, Bool) -> c`  
 e) `type Fun (a, c) (a, b) = (b, c)`  
 f) `data F = X Int | Y Float | Z X`  
 g) `data F = intfun Int`  
 h) `data F = Makro Int -> Int`  
 i) `type Val = Int | Bool`  
 j) `data X = X X X`

Př. 4.1.5 Uvažte datový typ `type Frac = (Int, Int)`, kde hodnota  $(a, b)$  představuje zlomek  $\frac{a}{b}$  (můžete předpokládat, že  $b \neq 0$ ). Napište funkci nad datovým typem `Frac`, která



- a) zjistí, jestli zadané dva zlomky představují stejné racionální číslo;  
 b) vrátí `True`, jestli zlomek představuje nezáporné číslo;  
 c) vypočítá součet dvou zlomků;  
 d) vypočítá rozdíl dvou zlomků;  
 e) vypočítá součin dvou zlomků;  
 f) vypočítá podíl dvou zlomků (ověřte, že druhý zlomek je nenulový);  
 g) převede zlomek do základního tvaru; *Doporučujeme*: zkuste si najít v dokumentaci něco o funkci `gcd`.

Když budete mít všechno implementované, tak upravte funkce tak, aby byl výsledek v základním tvaru.

Př. 4.1.6 V řetězci kaváren StarBugs prodávají jediný druh šálků kávy. Obyčejní zákazníci platí 13 λ za šálek kávy a každý desátý šálek mají zdarma. Pokud si kávu kupuje zaměstnanec, má navíc 15% slevu ze základní ceny. V případě, že si kávu kupuje student ve zkouškovém období, platí za každý šálek 1 λ. Napište funkci, která spočítá výslednou cenu v závislosti na typu zákazníka. Ale pozor! Slevový systém se často mění, aby zaujal lidi. Proto je potřeba navrhnout funkci dostatečně obecně, aby se nemusela vždy celá přepisovat.



- Napište funkci `commonPricing :: Int -> Float`, která na základě počtu vypitých šálků spočítá cenu pro běžného zákazníka.
- Napište funkci `employeeDiscount :: Float -> Float`, která aplikuje zaměstnaneckou slevu na cenu pro obyčejné zákazníky.
- Napište funkci `studentPricing :: Int -> Float`, která na základě počtu vypitých šálků spočítá cenu pro studenta.
- Definujte datový typ `PricingType`, který bude značit, zdali je nakupující obyčejný zákazník (`Common`), zaměstnanec řetězce (`Employee`) nebo student (`Student`).
- Implementujte funkci

```
computePrice :: PricingType -> Int -> (Int ->
 ↪ Float)
 -> (Float -> Float) -> (Int ->
 ↪ Float) -> Float
```


Ta podle typu zákazníka, počtu šálků a tří funkcí `cp`, `ed` a `sp` (common pricing, employee discount a student pricing) spočítá výslednou cenu za nakoupené šálky.

Při řešení se vám může hodit funkce `fromIntegral`. Více se o ní můžete dočíst v dokumentaci.

Příklady vstupů a odpovídajících výsledků:

```
computePrice Common 28 commonPricing employeeDiscount
 ↪ studentPricing
 ~>* 338
computePrice Employee 28 commonPricing employeeDiscount
 ↪ studentPricing
 ~>* 287.30002
computePrice Student 28 commonPricing employeeDiscount
 ↪ studentPricing
 ~>* 28
```

Následující příklad je rozšířením předchozí úlohy. Doporučujeme vrátit se k němu, pokud máte na konci cvičení čas.

- Př. 4.1.7**  Řetězec `StarBugs` z úlohy 4.1.6 se rozhodl rozšířit svůj systém slev. Ještě neví jak přesně, ale každá cena bude buď závislá na počtu koupených šálků, nebo na běžné ceně pro obyčejné zákazníky za daný počet šálků.

Upravte datový typ `PricingType` tak, aby nabízel možnosti `Common` (běžný zákazník), `Special (Int -> Float)` (speciální druh nacenění závislý na počtu káv) a `Discount (Float -> Float)` (slevový druh nacenění závislý na běžné ceně). Každá instance tohoto typu (krom `Common`) tak v sobě bude nést funkci pro výpočet správné ceny.


Dále je nezbytné změnit i funkci `computePrice`, a to tak, že její typ bude `PricingType -> Int -> (Int -> Float) -> Float`. Akceptuje druh zákazníka, počet šálků a funkci pro výpočet běžné ceny a vrací správnou cenu za příslušný počet šálků.

Jako poslední definujte konstanty `common`, `employee`, `student :: PricingType`, které reprezentují typy zákazníků ze cvičení 4.1.6.

Příklady volání a správných vyhodnocení:

```
computePrice common 28 commonPricing ~>* 338
computePrice employee 28 commonPricing ~>* 287.30002
computePrice student 28 commonPricing ~>* 28
```

## 4.2 Konstruktor *Maybe*

- Př. 4.2.1**  Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. Korektnost uvažujte pouze v kontextu standardně definovaných datových typů a funkcí (tj. těch z modulu `Prelude`), bez dalších hypotetických definic.

- `Maybe (Just 2)`
- `Just Just 2`
- `Just (Just 2)`

- d) `Maybe Nothing`
- e) `Nothing 3`
- f) `Just [Just 3]`
- g) `Just`

**Př. 4.2.2** Které ze zadaných výrazů jsou korektní? U korektních výrazů rozhodněte, jestli se jedná o hodnotu nebo o typ. U hodnot určete jejich typ a u typů uveďte příklady hodnot daného typu. S výrazem `a` pracujte jako s externě definovaným typem.



- a) `Maybe a`
- b) `Just a`
- c) `Just (\x -> x ^ 2)`
- d) `Just Just`
- e) `Just Just Just`

**Př. 4.2.3** Definujte funkci `divlist :: Integral a => [a] -> [a] -> [Maybe a]`, s využitím typového konstruktora `Maybe`, která celočíselně podělí dva celočíselné seznamy „po složkách“ a ošetří případy dělení nulou. Tedy například



```
divlist [12, 5, 7] [3, 0, 2] ~>* [Just 4, Nothing, Just 3]
divlist [12, 5, 7] [3, 1, 2, 5] ~>* [Just 4, Just 5, Just 3]
divlist [42, 42] [0] ~>* [Nothing]
```

**Př. 4.2.4** Napište funkci `addVars :: String -> String -> [(String, Integer)] -> Maybe Integer`, která dostane dva názvy proměnných a seznam přiřazující hodnoty proměnným a sečte hodnoty daných proměnných. Pokud se některá z proměnných v seznamu nenachází, pak vraťte `Nothing` (v opačném případě vraťte vhodně zabalený součet).



Pro vyhledání hodnoty proměnné použijte knihovní funkci `lookup :: Eq a => a -> [(a, b)] -> Maybe b`, která vyhledává podle svého prvního argumentu.

*Nápověda:* Zkuste nejprve vyhledat obě proměnné v seznamu proměnných a na výsledky těchto vyhledávání aplikovat vhodnou pomocnou funkci.

```
addVars "x" "y" [] ~>* Nothing
addVars "x" "y" [("x", 42)] ~>* Nothing
addVars "x" "y" [("a", 0), ("y", 42)] ~>* Nothing
addVars "a" "a" [("a", 0), ("y", 42)] ~>* Just 0
addVars "x" "y" [("a", 0), ("x", 12), ("y", 30)] ~>* Just 42
```

**Pan Fešák doporučuje:** Pokud si nejste jistí, co funkce `zip` přesně dělá, tak se podívejte do dokumentace.

**Př. 4.2.5** Napište funkci `mayZip :: [a] -> [b] -> [(Maybe a, Maybe b)]`, která je analogií funkce `zip`. Rozdílem je, že výsledný seznam má délku rovnou delšímu ze vstupních seznamů. Chybějící hodnoty jsou nahrazeny hodnotami `Nothing`.



**Př. 4.2.6** Uvažme, že máme následující definice typů:

4.2.summar



```
data Mark = A | B | C | D | E | F | X | S deriving (Eq, Show)
type StudentName = String
type CourseName = String
data StudentResult = StudentResult StudentName CourseName (Maybe
 ↪ Mark)
 deriving (Eq, Show)
```

Definujte funkci `summarize :: StudentResult -> String`, která pro studentský vý-

sledek vrátí řetězec následujícího tvaru:

- pro záznam se známkou bude tvaru "NAME has MARK from COURSE";
- v opačném případě to bude "NAME has no result from COURSE".

```
map summarize [StudentResult "Adam" "IA014" (Just A),
 StudentResult "Jan" "IV115" Nothing,
 StudentResult "Martin" "IA038" (Just X)]
 ~>* ["Adam has A from IA014",
 "Jan has no result from IV115",
 "Martin has X from IA038"]
```

*Poznámka:* Připomínáme existenci funkce `show :: Show a => a -> String`.

## 4.3 Rekurzivní datové typy

**Př. 4.3.1** Uvažme následující rekurzivní datový typ:

```
data Nat = Zero | Succ Nat deriving Show
```

- Jaké hodnoty má typ `Nat`?
- Jaký význam má dovětek `deriving Show`?
- Nadefinujte funkci `natToInt :: Nat -> Int`, která převede výraz typu `Nat` na číslo, které vyjadřuje počet použití hodnotového konstruktora `Succ` v daném výrazu.
- Jak byste pomocí datového typu `Nat` zapsali nekonečno?

**Př. 4.3.2** Uvažme následující definici typu `Expr`:

4.3.expr

```
data Expr = Con Double
 | Add Expr Expr | Sub Expr Expr
 | Mul Expr Expr | Div Expr Expr
```

- Uvedte výraz typu `Expr`, který představuje hodnotu 3.14.
- Definujte funkci `eval :: Expr -> Double`, která vrátí hodnotu daného výrazu.
- Ošetřete korektně dělení nulou pomocí funkce `evalMay :: Expr -> Maybe Double`.

**Př. 4.3.3** Rozšiřte definici z předchozího příkladu o nulární hodnotový konstruktore `Var`, který bude zastupovat proměnnou. Funkci `eval` upravte tak, aby jako první argument vzala hodnotu proměnné a vyhodnotila výraz z druhého argumentu pro dané ohodnocení proměnné.

\*  
\*\*

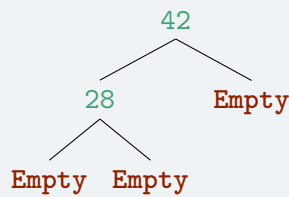
**Paní Bílá vysvětluje:** Binární strom (`BinTree a`) je struktura, která v každém svém uzlu `Node` udržuje hodnotu typu `a` a ukazatele na své dva potomky. Hodnotový konstruktore `Empty` reprezentuje prázdný strom. Používáme ho například na označení, že v daném směru uzel nemá potomka – `Node 1 Empty (Node 2 Empty Empty)` – uzel s hodnotou 1 nemá levého potomka, uzel s hodnotou 2 nemá ani jednoho potomka.

V následujících příkladech se využívá datová struktura

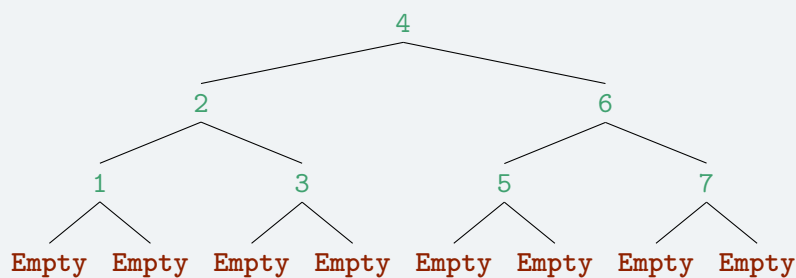
```
data BinTree a = Empty
 | Node a (BinTree a) (BinTree a)
 deriving Show
```

Následují příklady několika stromů: jejich zápis v Haskellu a jejich struktura.

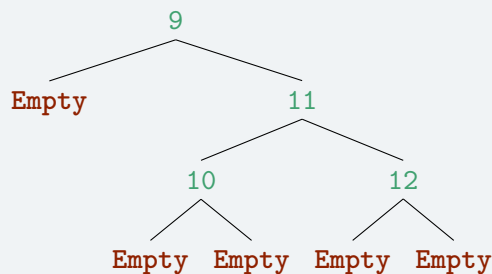
```
tree00 = Node 42 (Node 28 Empty Empty) Empty
```



```
tree01 = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
 (Node 6 (Node 5 Empty Empty) (Node 7 Empty Empty))
```



```
tree02 = Node 9 Empty (Node 11 (Node 10 Empty Empty)
 (Node 12 Empty Empty))
```



Prázdný strom (strom, který neuchovává žádnou hodnotu; podobně jako prázdný seznam) vypadá následovně:

```
emptyTree = Empty
```

```
Empty
```

#### Př. 4.3.4



- Nakreslete všechny tříuzlové stromy typu `BinTree ()` a zapište je pomocí hodnotových konstruktorů `Node` a `Empty`.
- Kolik existuje stromů typu `BinTree ()` s 0, 1, 2, 3, 4 nebo 5 uzly?
- Kolik existuje stromů typu `BinTree Bool` s 0, 1, 2, 3, 4 nebo 5 uzly?

**Pan Fešák doporučuje:** V příloženém souboru `04_data.hs` najdete taky předdefinované stromy, které můžete použít pro testování funkcí pracujících se strukturou `BinTree`.



**Př. 4.3.5** Pro datový typ `BinTree` a označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu. Definujte následující funkce nad binárními stromy:

4.3.tree

»=

- `treeSize :: BinTree a -> Int`, která spočítá počet uzlů ve stromě.
- `listTree :: BinTree a -> [a]`, která vrátí seznam hodnot, které jsou uloženy v uzlech vstupního stromu (na pořadí nezáleží).
- `height :: BinTree a -> Int`, která určí výšku stromu.
- `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě začínající v kořeni a vrátí ohodnocení na ní.

Pár příkladů vyhodnocení funkcí v této úloze:

```
treeSize Empty ~>* 0
treeSize tree01 ~>* 7
listTree tree01 ~>* [1, 2, 3, 4, 5, 6, 7] -- Jedno z možných řešení.
listTree tree02 ~>* [9, 10, 11, 12]
height tree02 ~>* 3
longestPath tree05 ~>* [100, 101, 102, 103, 104]
```

**Př. 4.3.6** Pro datový typ `BinTree` a označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.



- Definujte funkci `fullTree :: Int -> a -> BinTree a`, která pro volání `fullTree n v` vytvoří binární strom výšky `n`, ve kterém jsou všechny větve stejně dlouhé a všechny uzly jsou ohodnocené hodnotou `v`.
- Definujte funkci `treeZip :: BinTree a -> BinTree b -> BinTree (a, b)` jako analogii seznamové funkce `zip`. Výsledný strom tedy obsahuje pouze ty uzly, které jsou v obou vstupních stromech.

**Př. 4.3.7** Napište `treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)` jako analogii seznamové funkce `mayZip` z příkladu 4.2.5. Vrchol v novém stromu bude existovat právě tehdy, pokud existuje aspoň v jednom ze vstupních stromů.



**Př. 4.3.8** Uvažme datový typ `BinTree a`.



- Definujte funkci `isTreeBST :: (Ord a) => BinTree a -> Bool`, která se vyhodnotí na `True`, jestli bude její první argument validní binární vyhledávací strom (BST).
- Definujte funkci `searchBST :: (Ord a) => a -> BinTree a -> Bool`, která projde BST z druhého argumentu v smyslu binárního vyhledávání a vyhodnotí se na `True` v případě, že její první argument najde v uzlech při vyhledávání.

Můžete předpokládat, že vstupní datový typ `a` je uspořádaný lineárně.

## 4.4 Další příklady

**Př. 4.4.1** Uvažte typ stromů s vrcholy libovolné arity definovaný následovně:

»=

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

Definujte následující:

- a) funkci `roseTreeSize :: RoseTree a -> Int`, která spočítá počet uzlů ve stromě,  
 b) funkci `roseTreeSum :: Num a => RoseTree a -> a`, která sečte ohodnocení všech uzlů stromu,  
 c) funkci `roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b`, která bere funkci `a` a strom `a` a aplikuje danou funkci na hodnotu v každém uzlu:

```
roseTreeMap (+1) (RoseNode 0 [RoseNode 1 [], RoseNode
 ↪ 41 []])
 ~>* RoseNode 1 [RoseNode 2 [], RoseNode 42 []]
```

**Př. 4.4.2** Uvažme následující definici typu `LogicExpr`:

```
data LogicExpr = Pos | Neg
 | And LogicExpr LogicExpr
 | Or LogicExpr LogicExpr
 | Implies LogicExpr LogicExpr
 | Equiv LogicExpr LogicExpr
```

Definujte funkci `evalExpr :: LogicExpr -> Bool`, která vrátí hodnotu, na kterou se daný výraz vyhodnotí.

**Př. 4.4.3** Uvažujme rekurzivní datový typ `IntSet` definovaný takto:



```
data IntSet = SetNode Bool IntSet IntSet -- Node isEnd zero
 ↪ one
 | SetLeaf
 deriving Show
```

Ve stromě typu `IntSet` každá cesta z vrcholu jednoznačně určuje binární kód složený z čísel přechodů mezi otcem a synem (podle označení syna `one` respektive `zero`). Toho můžeme využít pro ukládání přirozených čísel do takového stromu. Strom typu `IntSet` obsahuje číslo  $n$  právě tehdy, pokud obsahuje cestu odpovídající binárnímu zápisu čísla  $n$ , a navíc poslední vrchol této cesty má nastavenou hodnotu `isEnd` na `True`.

Implementujte tyto funkce pro práci se strukturou `IntSet`:

- `insert :: IntSet -> Int -> IntSet` – obdrží strom typu `IntSet` a přirozené číslo  $n$  a navrátí strom obsahující číslo  $n$ .
- `find :: IntSet -> Int -> Bool` – obdrží strom typu `IntSet` a přirozené číslo  $n$  a vrátí `True` právě tehdy, pokud strom obsahuje  $n$ .
- `listSet :: IntSet -> [Int]` – obdrží strom typu `IntSet` a navrátí seznam čísel uložených v tomto stromě.

**Př. 4.4.4** Podobná stromová struktura jako v příkladu 4.4.3 by mohla být použita i pro udržování množiny řetězců nad libovolnou abecedou (například slova složená z písmen anglické abecedy nebo konečné posloupnosti celých čísel). Definujte datový typ `SeqSet` a sloužící pro uchovávání posloupností prvků typu `a`. Dále definujte obdoby funkcí ze cvičení 4.4.3:



- `insertSeq :: Eq a => SeqSet a -> [a] -> SeqSet a`
- `findSeq :: Eq a => SeqSet a -> [a] -> Bool`

\*  
\*\*

**Na konci cvičení byste měli zvládnout:**

- ▶ tvorbu vlastních datových typů;
- ▶ využívat datový typ `Maybe`;

- ▶ implementovat funkce na rekurzivních datových typech, a to především na strukturách typu strom.

# Řešení

Řeš. 4.η.1

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

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

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

Případně můžeme využít i instance `Ord`:

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

Řeš. 4.η.2 a) Příklady hodnot jsou:

```
Cube 1 2 3
Cylinder (-3) (1/2)
```

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

- b) Hodnotové konstruktory jsou umístěny jako první identifikátor ve výrazech oddělených svíslítky. Tedy v tomto případě to jsou `Cube`, `Cylinder`. Také hodnotový konstruktor začíná velkým písmenem.
- c) Typové konstruktory můžeme rozlišit na nově definované a na ty, které jsou jenom použité. Typový konstruktor je vždy umístěn jako první identifikátor za klíčovým slovem `data`, tedy v tomto případě `Object`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Double`.
- d) Funkci budeme definovat po částech. Pro každý možný tvar hodnoty typu `Object`, tj. pro každý typ tělesa definujeme samostatný řádek funkce. Poznamenejme, že je nutné použít závorky kolem argumentů funkcí, aby byl tento výraz považován jako jeden argument, ne za několik argumentů. K definici funkcí můžeme využít i konstantu `pi`, která je v Haskellu standardně dostupná.

```
volume :: Object -> Double
volume (Cube x y z) = x * y * z
volume (Cylinder r v) = pi * r * r * v

surface :: Object -> Double
surface (Cube x y z) = 2 * (x * y + x * z + y *
 ↪ z)
surface (Cylinder r v) = 2 * pi * r * (v + r)
```

Řeš. 4.η.3

```
data Contained a = NoValue | Single a | Pair a a
data Compare a = SameContainer
 | SameValue (Contained a)
 | DifferentContainer

cmpContained :: Eq a => Contained a
```

```

-> Maybe (Contained a)
-> Compare a
cmpContained NoValue (Just NoValue) =
 SameValue NoValue
cmpContained (Single x1) (Just (Single x2)) =
 if x1 == x2
 then SameValue (Single x1)
 else SameContainer
cmpContained (Pair x1 y1) (Just (Pair x2 y2)) =
 if x1 == x2 && y1 == y2
 then SameValue (Pair x1 y1)
 else SameContainer
cmpContained _ _ = DifferentContainer

```

Nezapomeňte, že hodnotový konstruktor lze vnímat jako funkci jako každou jinou. Proto je potřeba závorky psát jenom tam, kde je potřeba určit pořadí, v jakém se mají funkce aplikovat.

V případě parametrů je jen potřeba odlišit, co všechno tvoří jeden parametr.

- Řeš. 4.η.4
- Korektní typ s hodnotou například `Nothing` nebo `Just 'A'`.
  - Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu.
  - Korektní hodnota typu `Maybe String`.
  - Nekorektní výraz – hodnotový konstruktor `Just` očekává hodnotu, `Integer` je ovšem typ.
  - Korektní hodnota typu `Maybe (Maybe a)`.
  - Korektní hodnota typu `Num a => [Maybe a]`.
  - Korektní hodnota typu `Bool -> Maybe a -> Maybe a`.

Řeš. 4.η.5

```

safeDiv :: Integral a => a -> a -> Maybe a
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

```

- Řeš. 4.η.6 `Maybe`-hodnotu je třeba rozebrat pomocí vzorů a v závislosti na tom, zda jde o `Just` obsahující hodnotu, nebo o `Nothing`, vrátit buď obsaženou hodnotu, nebo výchozí hodnotu.

```

valOrDef :: Maybe a -> a -> a
valOrDef (Just x) _ = x
valOrDef Nothing d = d

```

Téměř tato funkce existuje i ve standardní knihovně a to v modulu `Data.Maybe` a jmenuje se `fromMaybe` – tyto dvě funkce se liší jen pořadím argumentů.

- Řeš. 4.1.1 a) Příklady hodnot jsou:

```
Steel
```

```
Warrior 100 Leather 5
```

```
Warrior (-42) Leather (-4)
```

Definice datových typů nevynucuje, aby celočíselné parametry byly nezáporné.

- Hodnotové konstruktory jsou v tomto případě `Steel`, `Leather` a `Warrior`.
- Typové konstruktory jsou v tomto případě `Armor` a `Warrior`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Integer`. `WeaponsPoints` a `HealthPoints` nejsou typové konstruktory, pouze typové aliasy.

d) Řešení může být následující:

```

attack :: Warrior -> Warrior -> Warrior
attack (Warrior _ _ hit) (Warrior hp Leather w) =
 Warrior (max 0 (hp - hit)) Leather w
attack (Warrior _ _ hit) (Warrior hp Steel w) =
 Warrior (max 0 (hp - hit `div` 2)) Steel w

```

Řeš. 4.1.2

```

data Jar = EmptyJar
 | Cucumbers
 | Jam String
 | Compote Int
 deriving (Show, Eq)

today :: Int
today = 2022

stale :: Jar -> Bool
stale EmptyJar = False
stale Cucumbers = False
stale (Jam _) = False
stale (Compote x) = today - x >= 10

```

Alternativní řešení s menším počtem vzorů a hlavně přehlednějším zápisem (víme, že výsledek pro první tři případy je stejný):

```

stale' :: Jar -> Bool
stale' (Compote x) = today - x >= 10
stale' _ = False

```

Řeš. 4.1.3

- Nulární typový konstruktore **X**, unární hodnotový konstruktore **Value**.
- Nulární typové konstruktory **M** a **N**, nulární hodnotové konstruktory **A**, **B**, **C**, **D**, unární hodnotové konstruktory **N** a **M**.
- Chybná deklarace: **Hah** je v seznamu použito jako typový konstruktore, jedná se však o hodnotový konstruktore.
- Nulární typový konstruktore **FNM**, ternární hodnotový konstruktore **T**.
- Vytváří se pouze typové synonymum (nulární).
- Nulární typový konstruktore **E**, unární hodnotový konstruktore **E**.

Řeš. 4.1.4

- Ok.
- Nok, typová proměnná **a** musí být argumentem konstruktore **Makro**.
- Nok, hodnotový konstruktore není možné použít vícekrát.
- Nok, typová proměnná **c** musí být argumentem konstruktore **Fun**.
- Nok, argumenty konstruktore **Fun** mohou být pouze typové proměnné, ne složitější typové výrazy (tj. není možné použít definici podle vzoru).
- Nok, **Z X** není korektní výraz, protože **X** je hodnotový konstruktore.
- Nok, každý hodnotový konstruktore musí začínat velkým písmenem.
- Nok, výraz je interpretován jako hodnotový konstruktore **Makro** se třemi argumenty: **Int**, **->** a **Int** – je nutné přidat závorke kolem **(Int -> Int)**.
- Nok, syntax výrazu je chybná: **type** musí mít na pravé straně pouze jednu možnost (jedná se o typové synonymum, ne o nový datový typ).
- Ok, typové i hodnotové konstruktory mají stejný název. Na pravé straně definice je **X** nejdrív binárním hodnotovým a pak dvakrát nulárním typovým konstruktorem. Jediná

plně definovaná hodnota tohoto typu je  $x = X \ x \ x$ .

- Řeš. 4.1.5** a) Matematická definice rovnosti zlomků nám říká, že  $\frac{a}{b} = \frac{c}{d} \Leftrightarrow a \cdot d = b \cdot c$ . Funkci pak už snadno postavíme na této rovnosti.

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

- b) Zde opět použijeme vestavěnou funkci `signum`.

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

- c) K řešení nám opět pomůže si nejdříve matematicky zapsat požadovaný výraz:  $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ .

```
fracplus :: Frac -> Frac -> Frac
fracplus (a, b) (c, d) = simplify (a * d + b * c, b * d)
```

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

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

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

- g) Pro úpravu do základního tvaru postačí vydělit čitatele i jmenovatele jejich největším společným jmenovatelem (můžeme použít vestavěnou funkci `gcd`, která pracuje i se zápornými čísly). Musíme si však dát pozor na znaménko – základním tvarem zlomku $\frac{-2}{-4}$ je $\frac{1}{2}$ a nikoliv $\frac{-1}{-2}$. To můžeme zajistit následovně: číslo ve jmenovateli základního tvaru budeme mít vždy kladné a znaménko přeneseme do čitatele (například pomocí vestavěné funkce `signum`).

```
simplify :: Frac -> Frac
simplify (a, b) = ((signum b) * (a `div` d), abs (b `div`
  ↪ d))
  where d = gcd a b
```

Poznámka: Haskell má vestavěný typ pro zlomky, `Rational`. Ten je reprezentován v podstatě stejným způsobem, tedy jako dvě hodnoty typu `Integer`. Nicméně nejedná se přímo o dvojice, ale typ `Rational` definuje hodnotový konstruktory `%`, tedy například zlomek $\frac{1}{4}$ zapíšeme jako `1 % 4`. Datový typ `Rational` je instancí mnoha typových tříd, mimo jiné `Num` a `Fractional`, proto s ním lze pracovat jako s jinými číselnými typy v Haskellu a používat operátory jako `(+)`, `(-)`, `(*)`, `(/)`.

Řeš. 4.1.6

```
commonPricing :: Int -> Float
commonPricing cups = fromIntegral (13 * pricingCups)
  where
    pricingCups = cups - cups `div` 10

employeeDiscount :: Float -> Float
employeeDiscount basePrice = basePrice * (1 - 0.15)

studentPricing :: Int -> Float
studentPricing cups = fromIntegral cups
```

```

data PricingType = Common | Employee | Student

computePrice :: PricingType -> Int -> (Int -> Float)
              -> (Float -> Float) -> (Int -> Float) -> Float
computePrice Common cups cp _ _ = cp cups
computePrice Employee cups cp ed _ = (ed . cp) cups
computePrice Student cups _ _ sp = sp cups

commonPricing :: Int -> Float
commonPricing cups = fromIntegral (13 * pricingCups)
  where
    pricingCups = cups - cups `div` 10

employeeDiscount :: Float -> Float
employeeDiscount basePrice = basePrice * (1 - 0.15)

studentPricing :: Int -> Float
studentPricing cups = fromIntegral cups

data PricingType' = Common'
                  | Special (Int -> Float)
                  | Discount (Float -> Float)

computePrice :: PricingType' -> Int -> (Int -> Float) -> Float
computePrice Common' cups cp = cp cups
computePrice (Special sp) cups _ = sp cups
computePrice (Discount dp) cups cp = dp (cp cups)

common :: PricingType'
common = Common'

employee :: PricingType'
employee = Discount employeeDiscount

student :: PricingType'
student = Special studentPricing

```

- Řeš. 4.2.1
- Nekorektní výraz – typový konstruktor **Maybe** aplikovaný na hodnotu.
 - Nekorektní výraz – hodnotový konstruktor **Just** je aplikovaný na příliš mnoho argumentů.
 - Korektní hodnota typu **Num a => Maybe (Maybe a)**.
 - Nekorektní výraz – typový konstruktor **Maybe** aplikovaný na hodnotu **Nothing**.
 - Nekorektní výraz – nulární hodnotový konstruktor **Nothing** nebere žádné argumenty.
 - Korektní hodnota typu **(Num a) => Maybe [Maybe a]**.
 - Korektní hodnota (funkce) typu **a -> Maybe a**.

- Řeš. 4.2.2
- Korektní typ s hodnotou například **Nothing**.
 - Nekorektní výraz – hodnotový konstruktor **Just** očekává hodnotu, **a** je ovšem typ.
 - Korektní hodnota typu **(Num a) => Maybe (a -> a)**.
 - Korektní hodnota (ne funkce) typu **Maybe (a -> Maybe a)**.

- e) Nekorektní výraz – implicitní závorky jsou (**Just Just**) **Just** a podle předchozího příkladu víme, že **Just Just :: Maybe (a -> Maybe a)**. Avšak tento výraz není funkcí (je to **Maybe** výraz – podstatný je vnější typový konstruktor), a proto ho nemůžeme aplikovat na hodnotu, jako by to byla funkce.

Řeš. 4.2.3

```
divlist :: Integral a => [a] -> [a] -> [Maybe a]
divlist = zipWith safeDiv
```

Funkce `safeDiv` je definovaná stejně jako v příkladu 4.η.5.

Řeš. 4.2.5

```
mayZip :: [a] -> [b] -> [(Maybe a, Maybe b)]
mayZip (a : xa) (b : xb) = (Just a, Just b) : (mayZip xa xb)
mayZip [] (b : xb) = (Nothing, Just b) : (mayZip [] xb)
mayZip (a : xa) [] = (Just a, Nothing) : (mayZip xa [])
mayZip [] [] = []
```

Řeš. 4.2.6

```
summarize :: StudentResult -> String
summarize (StudentResult name course res) =
  unwords [name, "has", showRes res, "from", course]
  where
    showRes (Just res) = show res
    showRes _ = "no result"
```

Řeš. 4.3.1 a) **Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero)), ...**

- b) Zajistí, že kompilátor deklaruje **Nat** jako instanci typové třídy **Show** (tj. typové třídy poskytující funkci `show`, která umožní převést hodnotu typu na jeho řetězcovou interpretaci) a na základě definice datového typu **Nat** automaticky definuje intuitivním způsobem funkci `show`, tj. např. `show (Succ (Succ Zero)) ~>* "Succ (Succ Zero)"`.

c)

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

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

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

Řeš. 4.3.2 a) **Con 3.14**

b)

```
eval :: Expr -> Double
eval (Con x) = x
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x - eval y
eval (Mul x y) = eval x * eval y
eval (Div x y) = eval x / eval y
```

- c) Pro získání hodnoty z výrazu tvaru **Just x** použijeme funkci `fromJust` a pro zjištění, zda hodnota existuje funkci `isJust`. Obě naleznete v modulu `Data.Maybe`, nebo si je můžete definovat sami:

```
fromJust :: Maybe a -> a
fromJust (Just x) = x
```

```
isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust _       = False
```

Pozor, mezi použitím `isJust x` a `x /= Nothing` je podstatný rozdíl v tom, že to druhé vyžaduje, aby typ zabalený uvnitř `Maybe` hodnoty byl porovnatelný.

Nyní již samotné řešení.

```
import Data.Maybe

evalMay :: Expr -> Maybe Double
evalMay (Con x) = Just x
evalMay (Add x y) = if isJust evx && isJust evy
  then Just (fromJust evx + fromJust evy)
  else Nothing
  where
    evx = evalMay x
    evy = evalMay y

-- vyhodnocovani pro konstruktory Sub a Mul je uplne
-- analogicke
-- vyhodnocovani Add, proto ho neuvadime

evalMay (Div x y) = if isJust evx && isJust evy
  && fromJust evy /= 0
  then Just (fromJust evx / fromJust evy)
  else Nothing
  where
    evx = evalMay x
    evy = evalMay y
```



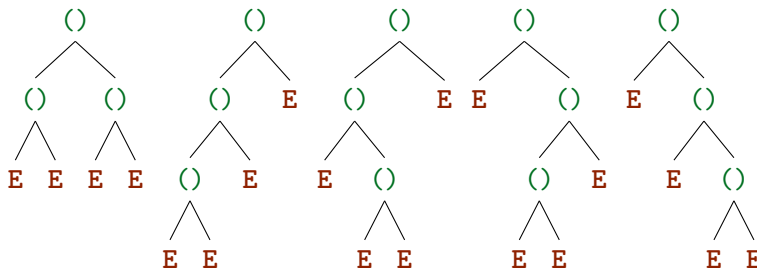
Tato úloha se dá o mnoho jednodušeji řešit s použitím tzv. monád (či alespoň Applicative). O nich se něco můžete dozvědět například na navazujících předmětech IB016 a IA014.

Řeš. 4.3.3

```
data Expr = Con Double      | Var
          | Add Expr Expr  | Sub Expr Expr
          | Mul Expr Expr  | Div Expr Expr

eval' :: Double -> Expr -> Double
eval' _ (Con x) = x
eval' v Var     = v
eval' v (Add x y) = eval' v x + eval' v y
eval' v (Sub x y) = eval' v x - eval' v y
eval' v (Mul x y) = eval' v x * eval' v y
eval' v (Div x y) = eval' v x / eval' v y
```

Řeš. 4.3.4 a) Jsou to tyto stromy:



```
tree1 = Node () (Node () Empty Empty) (Node ()
  ↪ Empty Empty)
tree2 = Node () (Node () (Node () Empty Empty)
  ↪ Empty) Empty
tree3 = Node () (Node () Empty (Node () Empty
  ↪ Empty)) Empty
tree4 = Node () Empty (Node () (Node () Empty
  ↪ Empty) Empty)
tree5 = Node () Empty (Node () Empty (Node () Empty
  ↪ Empty))
```

b) Necht' $\#_0(n)$ je počet stromů typu `BinTree ()`. Pak lze nahlédnout, že

$$\#_0(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{n-1} \#_0(i)\#_0(n-i-1) & \text{if } n > 0 \end{cases}$$

Pokud bychom chtěli znát konkrétní hodnoty, můžeme formuli přepsat

```
count 0 = 1
count n = sum $ map (\i -> count i * count (n - i -
  ↪ 1))
  [0 .. n - 1]
```

čímž lehce zjistíme, že `map count [0..5] ~> [1,1,2,5,14,42]`.

c) Pro `BinTree Bool` platí

$$\#_{\text{Bool}}(n) = 2^n \#_0(n)$$

Obecně pro `BinTree t` máme:

$$\#_t(n) = |t|^n \#_0(n),$$

kde $|t|$ je počet různých hodnot typu t . Hledané hodnoty $\#_{\text{Bool}}(n)$ jsou:

```
countT t n = t ^ n * count n
map (countT 2) [0, 1, 2, 3, 4, 5] ~>* [1, 2, 8, 40, 224, 1344]
```

Řeš. 4.3.5 a)

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

b)

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

c)

```
height :: BinTree a -> Int
height Empty = 0
```

```

height (Node _ l r) = 1 + max (height l) (height r)
d) longestPath :: BinTree a -> [a]
longestPath Empty = []
longestPath (Node v t1 t2) = if length p1 > length p2
    then v : p1
    else v : p2
    where
        p1 = longestPath t1
        p2 = longestPath t2

```

Řeš. 4.3.6 a)

```

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

```

b)

```

treeZip :: BinTree a -> BinTree b -> BinTree (a, b)
treeZip (Node x1 l1 r1) (Node x2 l2 r2) =
    Node (x1, x2) (treeZip l1 l2) (treeZip r1 r2)
treeZip _ _ = Empty

```

Řeš. 4.3.7

```

treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a,
    ↪ Maybe b)
treeMayZip (Node a l1 r1) (Node b l2 r2) =
    Node (Just a, Just b) (treeMayZip l1 l2) (treeMayZip r1
    ↪ r2)
treeMayZip (Node a l1 r1) Empty =
    Node (Just a, Nothing) (treeMayZip l1 Empty) (treeMayZip
    ↪ r1 Empty)
treeMayZip Empty (Node b l2 r2) =
    Node (Nothing, Just b) (treeMayZip Empty l2) (treeMayZip
    ↪ Empty r2)
treeMayZip Empty Empty = Empty

```

Řeš. 4.3.8 a)

```

-- Rozšíření množiny hodnot typu `a` o nekonečno
data MayInf a = Inf | NegInf | Fin a
    deriving (Eq)

instance (Ord a) => Ord (MayInf a) where
    _ <= Inf = True
    Inf <= _ = False
    NegInf <= _ = True
    _ <= NegInf = False
    (Fin a) <= (Fin b) = a <= b

isTreeBST :: (Ord a) => BinTree a -> Bool
isTreeBST = isTreeBST' NegInf Inf

isTreeBST' :: (Ord a) => MayInf a -> MayInf a ->
    ↪ BinTree a -> Bool
isTreeBST' _ _ Empty = True
isTreeBST' low high (Node v l r) = let v' = Fin v in

```

```

    low <= Fin v && Fin v <= high &&
    isTreeBST' v' high r &&
    isTreeBST' low v' l

-- alternativní definice
isTreeBST2 :: (Ord a) => BinTree a -> Bool
isTreeBST2 = isAscendingList . inorder

inorder :: BinTree a -> [a]
inorder Empty      = []
inorder (Node v l r) = inorder l ++ [v] ++ inorder r

isAscendingList :: (Ord a) => [a] -> Bool
isAscendingList [] = True
isAscendingList l = and $ zipWith (<=) l (tail l)

b) searchBST :: (Ord a) => a -> BinTree a -> Bool
searchBST _ Empty = False
searchBST k (Node v l r) = case compare k v of
    EQ -> True
    LT -> searchBST k l
    GT -> searchBST k r

```

Řeš. 4.4.1

```

roseTreeSize :: RoseTree a -> Int
roseTreeSize (RoseNode _ subtrees) =
    1 + sum (map roseTreeSize subtrees)

roseTreeSum :: Num a => RoseTree a -> a
roseTreeSum (RoseNode v subtrees) =
    v + sum (map roseTreeSum subtrees)

roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap f (RoseNode v subtrees) =
    RoseNode (f v) (map (roseTreeMap f) subtrees)

```

Řeš. 4.4.2

```

evalExpr :: LogicExpr -> Bool
evalExpr Pos      = True
evalExpr Neg      = False
evalExpr (And x y) = evalExpr x && evalExpr y
evalExpr (Or x y)  = evalExpr x || evalExpr y
evalExpr (Implies x y) = not (evalExpr x) || evalExpr y
evalExpr (Equiv x y) = evalExpr x == evalExpr y

```

Řeš. 4.4.3

```

data Bit = 0 | 1
    deriving Show

toBitString :: Int -> [Bit]
toBitString 0 = [0]
toBitString a = (if a `mod` 2 == 1 then I else O)
    : toBitString (a `div` 2)

```

```

fromBitString :: [Bit] -> Int
fromBitString (0 : xs) = 2 * fromBitString xs
fromBitString (1 : xs) = 1 + 2 * fromBitString xs
fromBitString []       = 0

insert :: IntSet -> Int -> IntSet
insert set num = insert' set (toBitString num)

insert' :: IntSet -> [Bit] -> IntSet
insert' SetLeaf []           = SetNode True SetLeaf SetLeaf
insert' (SetNode _ l r) [] = SetNode True l r
insert' SetLeaf (0 : bits) = SetNode False (insert' SetLeaf
  ↪ bits) SetLeaf
insert' SetLeaf (1 : bits) = SetNode False SetLeaf (insert'
  ↪ SetLeaf bits)
insert' (SetNode end l r) (0 : bits) = SetNode end (insert' l
  ↪ bits) r
insert' (SetNode end l r) (1 : bits) = SetNode end l (insert'
  ↪ r bits)

find :: IntSet -> Int -> Bool
find set num = find' set (toBitString num)

find' :: IntSet -> [Bit] -> Bool
find' (SetNode True _ _) [] = True
find' _ []                  = False
find' SetLeaf _             = False
find' (SetNode _ l _) (0 : xs) = find' l xs
find' (SetNode _ _ r) (1 : xs) = find' r xs

listSet :: IntSet -> [Int]
listSet set = listSet' set []

listSet' :: IntSet -> [Bit] -> [Int]
listSet' SetLeaf _ = []
listSet' (SetNode False l r) bits = listSet' l (0 : bits)
  ++ listSet' r (1 : bits)
listSet' (SetNode True l r) bits = fromBitString (reverse
  ↪ bits)
  : listSet' l (0 : bits) ++ listSet' r (1 : bits)

Řeš. 4.4.4
data SeqSet a = SeqNode Bool [(a, SeqSet a)]
  deriving Show

son :: (Eq a) => a -> [(a, SeqSet a)] -> SeqSet a
son a anc = getSon $ lookup a anc
  where
    getSon Nothing = SeqNode False []
    getSon (Just node) = node

```

```
insertSeq :: Eq a => SeqSet a -> [a] -> SeqSet a
insertSeq (SeqNode _ anc) [] = SeqNode True anc
insertSeq (SeqNode end anc) (a : xa) = let s = son a anc in
    SeqNode end ((a, insertSeq s xa) : filter ((a/=) . fst)
    ↪ anc)
```

```
findSeq :: Eq a => SeqSet a -> [a] -> Bool
findSeq (SeqNode end _ ) [] = end
findSeq (SeqNode _ anc) (a : xa) = findSeq (son a anc) xa
```

Přiložený kód

Pan Sazeč upozorňuje: Kopírování kódu ze souboru PDF nezachovává odsazení! Soubory jsou ale vloženy i jako přílohy tohoto dokumentu; s rozumným prohlížečem je můžete stáhnout kliknutím na název souboru, ať už zde, či v příslušných příkladech. Přílohy jsou k nalezení také ve studijních materiálech v ISu.

 04_data.hs

```
-- * * * * * --
type WeaponsPoints = Integer
type HealthPoints = Integer

data Armor = Leather | Steel -- kožené nebo ocelové brnění
           deriving (Eq, Show)

data Warrior = Warrior HealthPoints Armor WeaponsPoints
             deriving (Eq, Show)

attack :: Warrior -> Warrior -> Warrior
attack = undefined

warrior1 = Warrior 30 Leather 30
warrior2 = Warrior 20 Steel 25

-- * * * * * --

addVars :: String -> String -> [(String, Integer)] -> Maybe Integer
addVars = undefined

data Mark = A | B | C | D | E | F | X | S deriving (Eq, Show)
type StudentName = String
type CourseName = String
data StudentResult = StudentResult StudentName CourseName (Maybe Mark)
                   deriving (Eq, Show)

summarize :: StudentResult -> String
summarize = undefined

-- * * * * * --
data Nat = Zero | Succ Nat deriving Show

natToInt :: Nat -> Int
natToInt = undefined

-- * * * * * --
data Expr = Con Double
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
          deriving Show

eval :: Expr -> Double
eval = undefined
```



```

evalMay :: Expr -> Maybe Double
evalMay = undefined

-- * * * * *
data BinTree a = Empty
              | Node a (BinTree a) (BinTree a)
              deriving (Eq, Show)

tree00 :: BinTree Int
tree00 = Node 42 (Node 28 Empty Empty) Empty

tree01 :: BinTree Int
tree01 = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
          (Node 6 (Node 5 Empty Empty) (Node 7 Empty Empty))

tree02 :: BinTree Int
tree02 = Node 9 Empty (Node 11 (Node 10 Empty Empty)
                               (Node 12 Empty Empty))

tree03 :: BinTree Int
tree03 = Node 8 tree01 tree02

tree04 :: BinTree Int
tree04 = Node 4 (Node 2 Empty (Node 3 Empty Empty))
          (Node 6 (Node 5 Empty Empty) Empty)

tree05 :: BinTree Int
tree05 = Node 100 (Node 101 Empty
                    (Node 102 (Node 103 Empty
                               (Node 104 Empty Empty))
                              Empty))
                    (Node 99 (Node 98 Empty Empty)
                             (Node 98 Empty Empty))

emptyTree :: BinTree Int
emptyTree = Empty

-- * * * * *
treeSize :: BinTree a -> Int
treeSize = undefined

listTree :: BinTree a -> [a]
listTree = undefined

height :: BinTree a -> Int
height = undefined

longestPath :: BinTree a -> [a]
longestPath = undefined

-- * * * * *
fullTree :: Int -> a -> BinTree a
fullTree = undefined

treeZip :: BinTree a -> BinTree b -> BinTree (a, b)

```

```

treeZip = undefined

-- * * * * *
treeMayZip :: BinTree a -> BinTree b -> BinTree (Maybe a, Maybe b)
treeMayZip = undefined

-- * * * * *
isTreeBST :: (Ord a) => BinTree a -> Bool
isTreeBST = undefined

searchBST :: (Ord a) => a -> BinTree a -> Bool
searchBST = undefined

-- * * * * *
data RoseTree a = RoseNode a [RoseTree a]
                 deriving (Show, Read)

roseTreeSize :: RoseTree a -> Int
roseTreeSize = undefined

roseTreeSum :: Num a => RoseTree a -> a
roseTreeSum = undefined

roseTreeMap :: (a -> b) -> RoseTree a -> RoseTree b
roseTreeMap = undefined

-- * * * * *
data LogicExpr = Pos | Neg
               | And LogicExpr LogicExpr
               | Or LogicExpr LogicExpr
               | Implies LogicExpr LogicExpr
               | Equiv LogicExpr LogicExpr

evalExpr :: LogicExpr -> Bool
evalExpr = undefined

-- * * * * *
data IntSet = SetNode Bool IntSet IntSet -- Node isEnd zero one
            | SetLeaf
            deriving Show

insert :: IntSet -> Int -> IntSet
insert = undefined

find :: IntSet -> Int -> Bool
find = undefined

listSet :: IntSet -> [Int]
listSet = undefined

-- * * * * *

```