Cvičení 5

5.1 Vlastní datové typy

Příklad 5.1.1 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.

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

Příklad 5.1.2 Vytvořte nový datový typ `Jar` představující sklenici ve spíži. Každá sklenice je v jednom z následujících stavů:

- 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 nekazí (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).

Příklad 5.1.3 Identifikujte nově vytvořené typové a datové konstruktory a určete jejich aritu.

a) `data X = X`
b) `data A = X | Y String | Z Int Int`
c) `data B a = A | B a | C a`
d) `data C = D C`
e) `data E = E (E, E)`
f) `type String = [Char]`

Příklad 5.1.4 Identifikujte nově vytvořené typové a datové konstruktory a určete jejich aritu.

a) `data X = Value Int`
b) `data X a = V a`
c) `data X = Test Int [Int] X`
d) `data X = X`
e)
```
data M = A | B | N M
data N = C | D | M N
```
f) `data Test a = F [Test] a | M Int (Maybe String) deriving Read`
g) `data Ha = Hah Int Float [Hah]`
h) `data FMN = T (Int, Int) (Int -> Int) [Int]`
i) `data LInt = [Int]`

j) `type Fat = Float -> Float -> Float`

Příklad 5.1.5 Mějme následující definici:

```
data Teleso = Kvadr Float Float Float  -- a,b,c
            | Valec Float Float        -- r,v
            | Kuzel Float Float        -- r,v
            | Koule Float              -- r
```

a) Jaké hodnoty má typ `Teleso`?
b) Kolik je v definici použito datových konstruktorů a které to jsou?
c) Kolik je v definici použito typových konstruktorů a které to jsou?
d) Definujte funkce `objem` a `povrch`, které pro hodnoty uvedeného typu počítají požadované.

Příklad 5.1.6 Uvažme následující definici typu `Expr`:

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

a) Uveďte výraz typu `Expr`, který představuje hodnotu 3.14.
b) Definujte funkci `eval :: Expr -> Float`, která vrátí hodnotu daného výrazu.

Příklad 5.1.7 Rozšiřte definici z předchozího příkladu o nulární datový konstruktor `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.

Příklad 5.1.8 Uvažte datový typ `type Frac = (Int, Int)`, kde hodnota `(a, b)` představuje zlomek $\frac{a}{b}$ (můžete předpokládat $b \neq 0$). Napište funkci nad datovým typem `Frac`, která

a) převede zlomek do základního tvaru;
b) zjistí, jestli se zadané dva zlomky rovnají;
c) vrátí `True`, jestli zlomek představuje nezáporné číslo;
d) vypočítá součet dvou zlomků;
e) vypočítá rozdíl dvou zlomků;
f) vypočítá součin dvou zlomků;
g) vypočítá podíl dvou zlomků (ověřte, že druhý zlomek je nenulový);
h) vrátí aritmetický průměr zadaného seznamu zlomků (opět ve formě zlomku).

Ve všech případech vraťte výsledek v základním tvaru.

Příklad 5.1.9 Které deklarace datových typů jsou správné?

a) `data M a = M a`
b) `data MujBool = Bool`
c) `type MujBool = Bool`
d) `data N x = NVal (x -> x)`
e) `type F = Bool -> Bool`
f) `type Makro = a -> a`
g) `data M = N (x, x) | N Bool | O M`

h) `type Fun a = a -> (a, Bool) -> c`
i) `type Fun (a, c) (a, b) = (b, c)`
j) `type Discarder a b c d e = b`
k) `data F = X Int | Y Float | Z X`
l) `data F = X Int | Y Float | Z (X Int)`
m) `data F = intfun Int`
n) `data F = Makro Int -> Int`
o) `type Val = Int | Bool`
p) `data M = Value M`
q)

```
data X1 = M Int
data X2 = M Float X1 | None
```

r)

```
data Choice x y = GoodChoice x | BadChoice y
type GC x = GoodChoice x
```

s)

```
data M1 = M1Val M2 | E
data M2 = M2Val M1
```

t) `data X = X X X`

Příklad 5.1.10 Určete typy následujících hodnot:

```
data T a = F String a | D String Int [T a]
```

a) `F String`
b) `D "abc" 2 [F "1" 1, F "2" 10]`
c) `D "main" 10 [F "Ano" "hello.c", F "Nie" "hello.o"]`
d) `[D "n1" 0 [], D "n2" 1 [T "2"]]`

```
data A a = M (a, a) | N [a] [A a] Int | O
```

e) `M ('a', 'S')`
f) `O`
g) `N [] [O, M (3, 3)] 0`
h) `x = N [] (repeat x) 10`

```
data M = A | B | N M
data N = C | D | M N
```

i) `N (M (N A))`
j) `M (M C)`

```
data X a b = T (X a b) | U (X b a) | V a | W b
```

k) `V True`
l) `T`
m) `[U (V (Just 2)), T (V [2])]`
n) `T . U . V`
o) `T (V W)`

5.2 Konstruktor `Maybe`

Příklad 5.2.1 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.

a) `Maybe (Just a)`
b) `Maybe a`
c) `Just a`
d) `Just Just 2`
e) `Maybe Nothing`
f) `Just Nothing`
g) `Nothing 3`
h) `[Just 4, Just Nothing]`
i) `Just [Just 3]`
j) `Just [] :: Maybe [Maybe Int]`
k) `(Just 3, Just Nothing) :: (Maybe Int, Maybe a)`
l) `Maybe [a -> Just Char]`
m) `Just (\x -> x^2)`
n) `(Just (+1)) (Just 10)`
o) `\b matters -> if b then Nothing else matters`
p) `Just`
q) `Just Just`
r) `Just Just Just`
s) `Maybe Maybe`
t) `Maybe ((Num a) => Maybe (a, a))`

Příklad 5.2.2 S využitím typového konstruktoru `Maybe` definujte funkci `divlist :: Integral a => [a] -> [a] -> [Maybe a]`, která celočíselně podělí dva celočíselné seznamy „po složkách“, tj.

$$\texttt{divlist [}x_1\texttt{, ..., }x_n\texttt{] [}y_1\texttt{, ..., }y_n\texttt{]} \rightsquigarrow^* \texttt{[div }x_1\ y_1\texttt{, ..., div }x_n\ y_n\texttt{]}$$

`divlist [12, 5, 7] [3, 0, 2]` $\rightsquigarrow^*$ `[Just 4, Nothing, Just 3]`

a ošetří případy dělení nulou.

Příklad 5.2.3 Uvažte následující datový typ:

```
data MyMaybe a = MyNothing
               | MyJust a
               deriving (Show, Read, Eq)
```

Deklarujte typ `MyMaybe` jako instanci typové třídy `Ord`. Za jakých podmínek může být typ `MyMaybe a` instancí třídy `Ord`?

5.3 Rekurzivní datové typy

Příklad 5.3.1 Uvažme následující rekurzivní datový typ:
`data Nat = Zero | Succ Nat deriving Show`

a) Jaké hodnoty má typ `Nat`?
b) Jaký význam má dovětek `deriving Show`?
c) Redefinujte způsob zobrazení hodnot typu `Nat`.
d) Nadefinujte funkci `natToInt :: Nat -> Int`, která převede výraz typu `Nat` na číslo, které vyjadřuje počet použití datového konstruktoru `Succ` v daném výrazu.
e) Jak byste pomocí datového typu `Nat` zapsali nekonečno?

Příklad 5.3.2 Uvažme následující rekurzivní typ představující binární strom s ohodnocenými uzly:

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

a) Nakreslete všechny tříuzlové stromy typu `BinTree ()` a zapište je pomocí datových konstruktorů `Node` a `Empty`.
b) Kolik existuje stromů typu `BinTree ()` s 0, 1, 2, 3, 4 nebo 5 uzly?
c) Kolik existuje stromů typu `BinTree Bool` s 0, 1, 2, 3, 4 nebo 5 uzly?
d) Definujte funkci `size :: BinTree a -> Int`, která určí počet uzlů stromu.

Příklad 5.3.3 Uvažte následující rekurzivní datový typ představující binární strom s ohodnocenými uzly:

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

Definujte následující funkce nad binárními stromy:

a) `treeSize :: BinTree a -> Integer`, která spočítá počet uzlů ve stromě.
b) `listTree :: BinTree a -> [a]`, která převede všechny hodnoty uzlů ve stromu do seznamu.
c) `height :: BinTree a -> Int`, která určí výšku stromu.
d) `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě začínající v kořeni a vrátí ohodnocení na ní.

Příklad 5.3.4 Pro datový typ `BinTree a` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.

a) 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 ohodnocené hodnotou `v`.

b) 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.

Příklad 5.3.5 Uvažme datový typ `BinTree a`.

a) Definujte funkci `treeRepeat :: a -> BinTree a` jako analogii seznamové funkce `repeat`. Funkce tedy vytvoří nekonečný strom, který má zadanou hodnotu v každém uzlu.
b) Pomocí funkce `treeRepeat` vyjádřete nekonečný binární strom `nilTree`, který má v každém uzlu prázdný seznam.
c) Definujte funkci `treeIterate :: (a->a) -> (a->a) -> a -> BinTree a` jako analogii seznamové funkce `iterate`. Levý potomek každého uzlu bude mít hodnotu vzniklou aplikací první zadané funkce a pravý aplikací druhé zadané funkce.

Příklad 5.3.6 Deklarujte typ `BinTree a` jako instanci typové třídy `Eq`. Instanci si napište sami (tj. nepoužívejte klauzuli `deriving`).

Příklad 5.3.7 Definujte nějaký binární strom, který má nekonečnou hloubku (zkuste to udělat různými způsoby).

Příklad 5.3.8 Uvažme datový typ `BinTree a`.

a) Definujte funkci `isTreeBST :: (Ord a) => BinTree a -> Bool`, která se vyhodnotí na `True`, jestli bude její první argument validní binární vyhledávací strom.
b) 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.

Řešení

Řešení 5.1.1

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

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

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

Řešení 5.1.2

```
data Jar = EmptyJar
   | Cucumbers
   | Jam String
   | Compote Int
   deriving (Show, Eq)
today :: Int
today = 2014
stale :: Jar -> Bool
stale EmptyJar = False
stale Cucumbers = False
stale (Jam _) = False
stale (Compote x) = today - x  >= 10
```

Řešení 5.1.3

a) Nulární typový konstruktor `X`, nulární datový konstruktor `X`.
b) Nulární typový konstruktor `A`, nulární datový konstruktor `X`, unární datový konstruktor `Y`, binární datový konstruktor `Z`.
c) Unární typový konstruktor `B` (konkrétní typ pak může být například `B Int` nebo `B [String]`). Nulární datový konstruktor `A`, unární datové konstruktory `B` a `C`.
d) Nulární typový konstruktor `C`, unární datový konstruktor `D`.
e) Nulární typový konstruktor `E`, unární datový konstruktor `E`.
f) Definice (nulárního) typového aliasu `String`.

Řešení 5.1.4

a) Nulární typový konstruktor `X`, unární datový konstruktor `Value`.
b) Unární typový konstruktor `X`, unární datový konstruktor `V`.
c) Nulární typový konstruktor `X`, ternární datový konstruktor `Test`.
d) Nulární typový konstruktor `X`, nulární datový konstruktor `X`.

e) Nulární typové konstruktory `M` a `N`, nulární datové konstruktory `A`, `B`, `C`, `D`, unární datové konstruktory `N` a `M`.
f) Unární typový konstruktor `Test`, binární datové konstruktory `F` a `M`.
g) Chybná deklarace: `Hah` je v seznamu použito jako typový konstruktor, jedná se však o datový konstruktor.
h) Nulární typový konstruktor `FNM`, ternární datový konstruktor `T`.
i) Chybná deklarace: chybí datový konstruktor.
j) Vytváří se pouze typové synonymum (nulární).

Řešení 5.1.5

a) Příklady hodnot jsou:

```
Kvadr 1 2 3
Kvadr (-4) 3 4.3
Valec 3 (1/2)
Koule (sin 2)
```

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

b) Datové konstruktory jsou umístěny jako první identifikátor ve výrazech oddělených svislítky. Tedy v tomto případě to jsou `Kvadr, Valec, Kuzel, Koule`. Také datový 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 `Teleso`. Kromě toho je tady použit i existující typový konstruktor, konkrétně `Float`.

d) Funkci budeme definovat po částech. Pro každý možný tvar hodnoty typu `Teleso`, tj. pro každý typ tělesa definujeme funkci osobitě. Poznamenejme, že je nutné použít závorky kolem argumentů funkcí, aby byl tento výraz považován jako jeden argument, ne jako několik argumentů. K definici funkcí můžeme využít i konstantu `pi`, která je v Haskellu standardně dostupná.

```
objem :: Teleso -> Float
objem (Kvadr x y z) = x * y * z
objem (Valec r v)   = pi * r * r * v
...

povrch :: Teleso -> Float
povrch (Kvadr x y z) = 2 * (x * y + x * z + y * z)
povrch (Valec r v)   = 2 * pi * r * (v + r)
...
```

Řešení 5.1.6

a) `Con 3.14`
b)

```
eval :: Expr -> Float
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
```

Řešení 5.1.7

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

eval :: Float -> Expr -> Float
eval _ (Con x)   = x
eval v (Var) = v
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
```

Řešení 5.1.8

a) 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
```

b) 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 téhle rovnosti.

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

c) Zde opět efektivně použijeme vestavěnou funkci `signum`.

```
nonneg :: Frac -> Bool
nonneg (a,b) = signum (a*b) >= 0
```

d) K řešení nám opět pomůže nejdříve si 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)
```

e)

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

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

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

h) Pro výpočet průměru musíme nejdříve určit součet všech zlomků v seznamu. To zjistíme kombinací akumulační funkce `foldr1` a součtu zlomků. Následně součet vydělíme délkou seznamu – jestliže však chceme použít funkci na dělení zlomků, kterou jsme definovali dřív, musíme délku převést na zlomek.
```
fracmean :: [Frac] -> Frac
fracmean s = fracdiv (foldr1 fracplus s) (length s,1)
```

Poznámka: Haskell má vestavěný typ pro zlomky, `Rational`, ten je reprezentován v podstatě stejným způsobem, jako dvě hodnoty typu `Integer`. Nicméně nejedná se o dvojice, typ `Rational` definuje datový konstruktor `%`, 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 `(+)`, `(-)`, `(*)`, `(/)`.

Řešení 5.1.9

a) Ok, datové a typové konstruktory můžou mít stejné názvy.
b) Nok, chybí datový konstruktor.
c) Ok, jednoduché typové synonymum.
d) Ok.
e) Ok.
f) Nok, typová proměnná `a` musí být argumentem konstruktoru `Makro`.
g) Nok, datový konstruktor není možné použít vícekrát.
h) Nok, typová proměnná `c` musí být argumentem konstruktoru `Fun`.
i) Nok, argumenty konstruktoru `Fun` mohou být pouze typové proměnné, ne složitější typové výrazy (tj. není možné použít definici podle vzoru).
j) Ok, typové synonymum nemusí být lineární ve svých argumentech (nemusí být každý použit právě jednou).
k) Nok, `Z X` není korektní výraz, protože `X` je datový konstruktor.
l) Nok, v argumentech datových konstruktorů se při deklaraci mohou vyskytovat pouze typy, ne datové konstruktory.
m) Nok, každý datový konstruktor musí začínat velkým písmenem.
n) Nok, výraz je interpretován jako datový konstruktor `Makro` se třemi argumenty: `Int`, `->` a `Int` – je nutné přidat závorky kolem `(Int -> Int)`.
o) Nok, syntax výrazu je chybná: `type` musí mít na pravé straně pouze jednu možnost (jedná se o typové synonymum, ne o nový datový typ).
p) Ok, i když neexistuje žádná konečná úplně definovaná hodnota. Hodnotou tohoto typu je například `x = Value x`.
q) Nok, není možné použít stejný datový konstruktor ve více typech.

r) Nok, chyba je v definici `type`: `GoodChoice` je datový konstruktor, tj. není možné, aby výsledkem byla jenom část typu (hodnoty typu `Choice t1 t2` vytvořené pomocí datového konstruktoru `GoodChoice`).
s) Ok, nepřímo rekurzivní datový typ je v pořádku.
t) Ok, typový i datové konstruktory mají stejný název. Na pravé straně definice je `X` nejdřív binárním datovým a pak dvakrát nulárním typovým konstruktorem. Jediná plně definovaná hodnota tohoto typu je `x = X x x`.

Řešení 5.1.10

a) Chybná hodnota, `String` je typ a není možné na něj aplikovat datový konstruktor.
b) `Num a => T a`
c) `T String`
d) Chybná hodnota, `T` není datový konstruktor.
e) `A Char`
f) `A a`
g) `Num a => A a`
h) `x :: A a`
i) Chybná hodnota, `N A :: M, M :: N -> N`.
j) `N`
k) `X Bool a`
l) `X a b -> X a b`
m) `(Num a, Num b) => [X [b] (Maybe a)]`
n) `a -> X b a`
o) `X (a -> X b a) c`

Řešení 5.2.1

a) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu.
b) Korektní typ s hodnotou například `Just 0`.
c) Korektní hodnota typu `Maybe t` v případě, že `a` je korektní hodnota (definovaná externě).
d) Nekorektní výraz – datový konstruktor `Just` je aplikovaný na příliš mnoho argumentů. Pro úplnost dodejme, že výraz `Just (Just 2)` by byl korektní hodnotou typu `Num a => Maybe (Maybe a)`.
e) Nekorektní výraz – typový konstruktor `Maybe` aplikovaný na hodnotu `Nothing`.
f) Korektní hodnota typu `Maybe (Maybe a)`.
g) Nekorektní výraz – nulární datový konstruktor `Nothing` nebere žádné argumenty.
h) Nekorektní výraz – jedna hodnota je typu `(Num a) => Maybe a`, druhá typu `Maybe (Maybe b)`.
i) Korektní hodnota typu `(Num a) => Maybe [Maybe a]`.
j) Korektní hodnota, typ je uvedený v zadání.
k) Nekorektní výraz – `Just Nothing` je typu `Maybe (Maybe a)`. Typ uvedený v zadání je všeobecnější než skutečný nejvšeobecnější typ výrazu.
l) Nekorektní výraz – typový konstruktor `Maybe` obsahující uvnitř datový konstruktor `Just`. Dodejme, že `Maybe [a -> Maybe Char]` by byl korektní typ.
m) Korektní hodnota typu `(Num a) => Maybe (a -> a)`.
n) Nekorektní výraz – hodnota typu `(Num a) => Maybe (a -> a)` (která není funkcí) je aplikována na hodnotu typu `(Num b) => Maybe b`.

o) Korektní hodnota typu `Bool -> Maybe a -> Maybe a`.
p) Korektní hodnota (funkce) typu `a -> Maybe a`.
q) Korektní hodnota (ne funkce) typu `Maybe (a -> Maybe a)`.
r) 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.
s) Nekorektní výraz – důvody jsou poněkud složitější. Typový konstruktor `Maybe` je druhu `* -> *`, tedy akceptuje typy druhu `*` a vrací typ s druhem `*`. Argument však nemá správný druh (substituci jako u typů není možné použít, druhy nejsou polymorfní).
t) Nekorektní výraz – typový kontext se udává pro celý typ, nikdy nesmí být zanořený uvnitř typu.

Řešení 5.2.2

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

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

Řešení 5.2.3 Nejprve vyřešme podmínku, kdy může taková instance existovat. Jinak řečeno, jaké podmínky jsou kladeny na datový typ `a`. Když budeme chtít porovnávat hodnoty typu `MyMaybe a`, budeme potřebovat porovnávat i hodnoty typu `a`. To tedy znamená, že `a` musí být instancí třídy `Ord`.

Ještě musíme dodefinovat nové uspořádání. Snad nejpřirozenějším (ale ne jediným možným!) uspořádáním je takové, které považuje `MyNothing` za nejmenší prvek a všechny hodnoty tvaru `MyJust x` za větší než `MyNothing`, přičemž uspořádání na hodnotách tvaru `MyJust x` je převzato z typu `a`.

Abychom mohli definovat typ jako instanci typové třídy, musíme k tomu definovat minimální množinu funkcí, která je pro danou třídu potřebná. Pro třídu `Ord` je to například množina {`(<=)`}. Ve výsledku tedy instanciace může vypadat následovně:

```
instance Ord a => Ord (MyMaybe a) where
    MyNothing <= _         = True
    MyJust x  <= MyJust y = x <= y
    _         <= _         = False
```

Řešení 5.3.1

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))` $\rightsquigarrow^*$ `"Succ (Succ Zero)"`.

c) Využijeme například analogii s Peanovými čísly – přirozenými čísly definovanými pomocí nuly a funkce následníka. Datový konstruktor `Zero` odpovídá nule, budete tedy psát `"0"`. Datový konstruktor `Succ` pak představuje přičtení jedničky, budeme tedy psát `"1+"`. Definice instance pak může vypadat třeba následovně:

```
instance Show Nat where
  show Zero = "0"
  show (Succ x) = "1+" ++ show x
```

Poznamenejme ještě, že pokud definujeme svoji vlastní instanci, klauzuli `deriving Show` musíme z definice typu odstranit.

d)

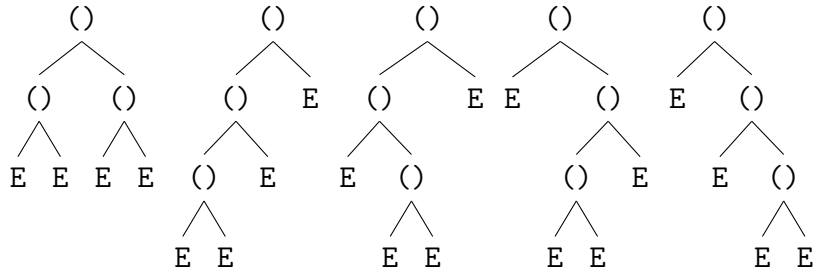
```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Succ x) = 1 + natToInt x
```

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

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

Řešení 5.3.2

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) Nechť $\#_{()}(n)$ je počet stromů typu `BinTree ()`. Pak lze nahlédnout, že

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

c)

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

Obecně pro `BinTree` t máme:

$$\#_t(n) = |t|^n \#_{()}(n),$$

kde $|t|$ je počet různých hodnot typu t.

d) Budeme postupovat rekurzivně vůči struktuře stromu. Strom může být buď tvaru `Empty` nebo tvaru `Node x l r`. V prvním případě je počet uzlů 0. V druhém případě je počet uzlů součtem počtu uzlů podstromů `l` a `r` plus jedna (za uzel samotný). Zapsáno vypadá definice následovně:

```
size :: BinTree a -> Int
size Empty        = 0
size (Node x l r) = 1 + size l + size r
```

Řešení 5.3.3

a)

```
treeSize :: BinTree a -> Integer
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 x 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
```

Řešení 5.3.4

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
```

Řešení 5.3.5

a)

```
treeRepeat :: a -> BinTree a
treeRepeat x = Node x (treeRepeat x) (treeRepeat x)
```

b)

```
nilTree :: BinTree [a]
nilTree = treeRepeat []
```

c)

```
treeIterate :: (a->a) -> (a->a) -> a -> BinTree a
treeIterate f g x =
    Node x (treeIterate f g (f x)) (treeIterate f g (g x))
```

Řešení 5.3.6

```
instance Eq a => Eq (BinTree a) where
    Empty          == Empty          = True
    Node x1 l1 r1 == Node x2 l2 r2 =
        x1 == x2 && l1 == l2 && r1 == r2
    _              == _              = False
```

Poslední řádek nelze vynechat – pokrývá porovnávání prázdného a neprázdného stromu.

Řešení 5.3.7

```
bt1 = Node 0 bt1 bt1
bt2 n = Node n sub sub where sub = bt2 (n + 1)
bt3 = Node "strom" bt3 Empty
bt4 = iterate (Node 0 Empty) Empty
bt5 = until (const False) (Node 0 Empty) Empty
```

Poznamenejme, že `bt4` není úplně přesné řešení, protože je to seznam konečných binárních stromů, kterého teoretickým posledním prvkem je požadovaný strom. Funkce `until` je standardně definována v Prelude následovně:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x
    | p x       = x
    | otherwise = until p f (f x)
```

Řešení 5.3.8

a)

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


AscendingList :: (Ord a) => [a] -> Bool
AscendingList [] = True
```

```
AscendingList [_] = True
AscendingList (x:y:xs) = x <= y && isAscendingList (y:xs)

 alternativní definice
AscendingList2 :: (Ord a) => [a] -> Bool
AscendingList2 [] = True
stinline{isAscendingList2 l = and $ zipWith (<=) l (tail l)}

TreeBST :: (Ord a) => BinTree a -> Bool
TreeBST = isAscendingList . inorder
```

b)

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