4.2 Typy zdefiniowane

Listy są generowane przez [] i (:)
czyli

data [α] = [] |
(:) α [α]

[] i (:) nazywają się generatory (typu [α]).

$$\begin{aligned} \text{data } T\ \alpha_1 \dots \alpha_k = \ & C_1\ \tau_{11} \dots \tau_{1m_1}\ | \\ & C_2\ \tau_{21} \dots \tau_{2m_2}\ | \\ & \vdots \\ & C_n\ \tau_{n1} \dots \tau_{nm_n} \end{aligned}$$

(i) enumeration types ($k = 0 = m_i$)

data Boole = True | False
data Dni = Po | Wt | .... | So | Nie

weekend So = True
weekend Nie = True
weekend _ = False

↳ weekend :: Dni → Boole

(ii) product types (n = 1)

```
type Nazwisko = String
type Wiek = Int
data Plec = M | K

data Osoba = O Nazwicko Plec Wiek
```

↳ O "Nowak" K 18 :: Osoba

bo O :: Nazwisko → Plec → Wiek → Osoba

```
kobieta (O _ K _) = True
kobieta     _     = False
```

czyli kobieta :: Osoba → Boole

(iii) typy polymorficzne (k ≥ 1)

```
data Union a b = L a | R b
```

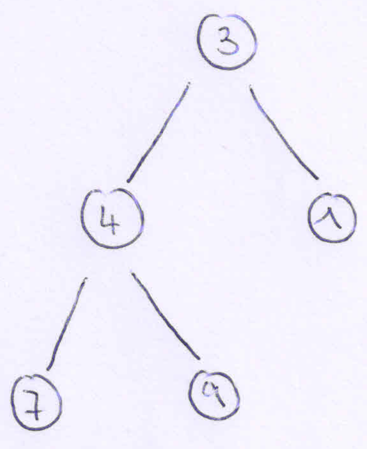
obiekty typu [Union a b], to listy "zawiesające" elementy typu a i b.

[L 1, R "a", L 8] :: [Union Int Char]

```
data Maybe a = Nothing | Just a

    lost [] = Nothing
    lost [x] = Just x
    lost (x:xs) = last xs

    sum [] = 0
    sum (Nothing:xs) = sum xs
    sum ((Just x): xs) = x + sum xs

  lost :: [α] -> Maybe α
  sum :: [Maybe Int] -> Int
```

(iv) recursive types

```
  data Tree a = Leaf a | Node a (Tree a) (Tree a)
```




```
t =
  Node 3
      (Node 4 (Leaf 7)(Leaf 9))
      (Leaf 1)
```

```
depth (Leaf x) = 1
depth (Node x l r) = 1 + max (depth l) (depth r)

sum (Leaf x) = x
sum (Node x l r) = x + (sum l) + (sum r)

depth :: Tree α -> Int
sum :: Tree Int -> Int

mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node x l r) =
        Node (f x) (mapTree f l) (mapTree f r)
```

ale: mapTree square t ↯

```
showTree (Leaf x) = x
showTree (Node x l r) =
        "<" ++ (showTree l) ++ "|" ++ x ++ "|"
            ++ (showTree r) ++ ">"

showTree (map (\x -> [x,x])
                (Node 1 (Leaf 3) (Leaf 4)))

"=" Node [1,1] (Leaf [3,3]) (Leaf [4,4])
```

↳ derive Show

4.3 Klasy typów

Problem: $1 + 1 :: Int$

$1.0 + 1.0 :: Real$

różne funkcje: $+_{\mathbb{Z}}, +_{\mathbb{R}}$

↳ overloading

w Haskellu: klasa typów

- kolekcja typów, dla których pewien zbiór operacji (funkcji) jest zdefiniowany – "interface"
- konkretny typ może używać operacje (funkcje), jeżeli jest elementem klasy.

```
>: type (+)
(Num α) => α -> α -> α
```

czyli

$\forall \alpha \in Num \quad (+) :: \alpha \to \alpha \to \alpha$

n.p.

$Int \in Num$, więc $(+) :: Int \to Int \to Int$

Przykład:

```
class (Eq a, Show a) => Num a where
    (+) :: a -> a -> a
    ⋮
```

czyli Num ⊆ Eq i Num ⊆ Show

```
class Eq a where
    (==), (\=) :: a -> a -> Bool
    x \= y = not (x == y)
```

↳ (==) :: Eq a => a -> a -> Bool

```
instance Eq Int where
    x == y = x 'integerEq' y

instance Eq a => Eq [a] where
    [] == [] = True
    [] == (x:xs) = False
    (x:xs) == [] = False
    (x:xs) = (y:ys) = x == y & xs == ys

instance Eq [a] where
    (==) = length
```

```
class Show a where
  show :: a -> String
```

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
> (Leaf 3)  ↯
```

```
instance (Show a) => Show (Tree a) where
    show (Leaf x) = show x
    show (Node l r) =
            "<" ++ show l ++ "|" ++ show r ++ ">"
```

```
> Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
<1|<2|3>>
```

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
                                    deriving Show
```

```
> Node (Leaf 5) (Leaf 7)
Node (Leaf 5) (Leaf 7)
```

Przykład:

Klasy z konstruktorami typów.

```
class Functor c where
  cmap :: (a -> b) -> c a -> c b

instance Functor [] where
  cmap f [] = []
  cmap f (x:xs) = (f x):(cmap f xs)

instance Functor Tree where
  cmap f (Leaf x) = Leaf (f x)
  cmap f (Node l r) =
              Node (cmap f l)(cmap f r)
```

Przykład:

kolejki → (21)

Przykład:

size → (22)

```haskell
class Queue q where

  isEmptyQ :: q a -> Bool
  addQ :: a -> q a -> q a
  delQ :: q a -> q a
  headQ :: q a -> a

  showQ :: (Show a) => q a -> String
  showQ qu = if isEmptyQ qu then "|"
             else (show (headQ qu)) ++ "-" ++ (showQ (delQ qu))


data Queue1 a = EmptyQ | AddQ a (Queue1 a)

instance Queue (Queue1) where

  isEmptyQ EmptyQ = True
  isEmptyQ _      = False

  addQ x q = AddQ x q

  delQ EmptyQ = error "delQ with emtpy queue"
  delQ (AddQ x q) = q

  headQ EmptyQ = error "delQ with emtpy queue"
  headQ (AddQ x q) = x

instance (Show a) => Show (Queue1 a) where
  show = showQ

qu1 = AddQ 1 (AddQ 2 (AddQ 3 EmptyQ))


data Queue2 a = Qu [a]

instance Queue (Queue2) where

  isEmptyQ (Qu [])  = True
  isEmptyQ _        = False

  addQ x (Qu xs)    = Qu (xs++[x])

  delQ (Qu [])      = error "delQ with emtpy queue"
  delQ (Qu (x:xs))  = Qu xs

  headQ (Qu [])     = error "delQ with emtpy queue"
  headQ (Qu (x:xs)) = x

instance (Show a) => Show (Queue2 a) where
  show qu = if isEmptyQ qu then "||"
            else (show (headQ qu)) ++ "+" ++ (show (delQ qu))

qu2 = Qu [1,2,3]
```

```
data Tree1 a     =  Nil | Node1 a (Tree1 a) (Tree1 a)

data Tree2 a b   =  Leaf2 b | Node2 a b (Tree2 a b) (Tree2 a b)

data Tree3       =  Leaf3 String | Node3 String Tree3 Tree3


sizeTree1 Nil                = 0
sizeTree1 (Node1 x l r)      = 1 + sizeTree1 l + sizeTree1 r

sizeTree2 (Leaf2 x)          = 1
sizeTree2 (Node2 x y l r)    = 2 + sizeTree2 l + sizeTree2 r

sizeTree3 (Leaf3 s)          = length s
sizeTree3 (Node3 s l r)      = length s + sizeTree3 l + sizeTree3 r

sizeList l                   = length l


class Size a where size :: a -> Int

instance Size (Tree1 a) where
  size Nil           = 0
  size (Node1 x l r) = 1 + size l + size r

instance Size (Tree2 a) where
  size (Leaf2 x)     = 0
  size (Node2 x l r) = 2 + size l + size r

instance Size (Tree3) where
  size (Leaf3 s)     = length s
  size (Node3 s l r) = length s + size l + size r

instance Size ([a]) where
  size = length


> size (Node1 3 (Leaf1 4) (Leaf1 1))
  3
> size (Leaf2 "abcd")
  1
> size (Leaf3 "abcd")
  4
> size [True,False,True]
  3
```