Typy definiowane przez użytkownika

Synonimy typów

Synonimy są skrótami dla już istniejących typów

Równoważne:

1.
```haskell
roots :: (Float, Float, Float) -> (Float, Float)
```

2.
```haskell
type Poly2 = (Float, Float, Float)
type Root2 = (Float, Float)

roots :: Poly2 -> Root2
```

Uwaga

Nazwy typów danych (i synonimów) muszą zaczynać się dużą literą

Typy użytkownika

```
data Polynom = Poly Float Float Float

Polynom  <-- nazwa typu
Poly     <-- nazwa konstruktora (funkcja)
Float    <-- typ 1go, 2go i 3go argumentu Poly

p :: Polynom
p = Poly 1.0 2.0 (-1.0)
```

Uwaga

Nazwa konstruktora może być taka sama jak nazwa typu

```
data Poly = Poly Float Float Float
```

Typy użytkownika a dopasowanie wzorca

```
data Polynom = Poly Float Float Float

roots :: (Float, Float, Float) -> (Float, Float)
roots (a, b, c) = ...

roots' :: Polynom -> (Float, Float)
roots' (Poly a b c) = ...
```

Przykład

```
data PointType = Point Float Float

p = Point 1 2

xPoint (Point x y) = x

ghci> xPoint p
1.0

data LineType = Line PointType PointType

dist (Line p1 p2) = sqrt ((xPoint p1 - xPoint p2)^2
                         + (yPoint p1 - yPoint p2)^2)

dist' (Line (Point x1 y1) (Point x2 y2)) =
                    sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

Przykład

```
firstQuad []                 = True
firstQuad ((Point x y):ps) = (x >= 0) && (y >= 0) &&
                              (firstQuad ps)

ghci> firstQuad [Point 1 2, Point 3 2, Point (-1) 1]
False
```

Alternatywne konstruktory

```
data PointType = Point Float Float

data Shape = Rectangle PointType PointType |
             Circle PointType Float |
             Triangle PointType PointType PointType

r = Rectangle (Point 2 4) (Point 8.5 2)
c = Circle (Point 1 1.5) 5.5
t = Triangle (Point 0 0) (Point 4.5 6) (Point 9 0)
```

```haskell
area :: Shape -> Float

area (Rectangle p1 p2) =
    abs (xPoint p1 - xPoint p2) *
    abs (yPoint p1 - yPoint p2)

area (Cirlce _ r) = pi * r^2

area (Triangle p1 p2 p3) =
    sqrt (h * (h - a) * (h - b) * (h - c))
    where
      h = (a + b + c) / 2.0
      a = dist p1 p2
      b = dist p1 p3
      c = dist p2 p3
      dist (Point x1 y1) (Point x2 y2)
        = sqrt ((x1 - x2)^2 * (y1 - y2)^2)
```

Konstruktory bezargumentowe

```
data Day = Mon | Tue | Wed | Thu | Fri |
           Sat | Sun

nameOfDay :: Day -> String
nameOfDay d = case d of
                Mon -> "Poniedzialek"
                Tue -> "Wtorek"
                Wed -> "Sroda"
                Thu -> "Czwartek"
                Fri -> "Piatek"
                Sat -> "Sobota"
                Sun -> "Niedziela"

ghci> nameOfDay Fri
"Piatek"
```

Typy parametryzowane

1.
```haskell
data PairType a = Pair a a

p = Pair 2 5 :: PairType Int

fstPair :: PairType a -> a
fstPair (Pair x _) = x

ghci> fstPair p
2
```

2.
```
data PairType a b = Pair a b

p = Pair 1 'a' :: PairType Int Char

sndPair :: PairType a b -> b
sndPair (Pair _ y) = y

ghci> sndPair p
'a'
```

Uwaga

Parametryzowane mogą być również synonimy typów, np.

```
type List a = [a]

l = ['a', 'b', 'c'] :: List Char
```

Typ *Maybe*

```
data Maybe a = Nothing | Just a

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] -> Maybe a
safehead []     = Nothing
safehead (x:xs) = Just x
```

```
ghci> safediv 3 2
Just 1
ghci> safediv 3 0
Nothing
ghci> safehead "haskell"
Just 'h'
ghci> safehead []
Nothing
```

Typ *Either*

```
data Either a b = Left a | Right b

foo :: Bool -> Either Char String
foo x = case x of
          True -> Left 'a'
          _    -> Right "a"

ghci> foo True
Left 'a'
ghci> foo False
Right "a"
```

Typy rekurencyjne

```
-- Liczba naturalna to "zero" lub jej następnik

data Nat = Zero | Succ Nat

n  = Zero
n1 = Succ Zero
n2 = Succ (Succ Zero)

add :: Nat -> Nat -> Nat
add m Zero = m
add m (Succ n) = Succ (add m n)

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

ghci> nat2int (add n1 n2)
3
```

Przykład – lista

```
Lista jest pusta, albo składa się z głowy i listy

data List a = Empty | Cons a (List a)

l :: List Int
l = Cons 12 (Cons 8 (Cons 10 Empty))

len :: List a -> Int
len Empty        = 0
len (Cons _ xs) = 1 + len xs

ghci> len l
3
```

Przykład – drzewo binarne

Drzewo binarne jest puste, albo składa się z wartości i dwóch poddrzew

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)

t :: Tree Int
t = Node 5 (Node 3 (Node 8 Empty Empty)
                   (Node 1 Empty Empty))
           (Node 4 Empty
                   (Node 6 Empty Empty))

depth :: Tree a -> Int
depth Empty        = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)

ghci> depth t
3
```

```
      5
     / \
    3   4
   / \   \
  8   1   6
```

Przechodzenie drzewa

Sposoby przechodzenia drzewa w głąb:

- *preorder* – wierzchołek zostaje odwiedzony zanim odwiedzone zostaną jego poddrzewa
- *inorder* – wierzchołek zostaje odwiedzony po odwiedzeniu lewego i przed odwiedzeniem jego prawego poddrzewa
- *postorder* – wierzchołek zostaje odwiedzony po odwiedzeniu jego lewego i prawego poddrzewa

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)

preorder :: Tree a -> [a]
preorder Empty        = []
preorder (Node a l r) = [a] ++ preorder l ++ preorder r

inorder Empty        = []
inorder (Node a l r) = inorder l ++ [a] ++ inorder r

postorder Empty        = []
postorder (Node a l r) = postorder l ++ postorder r ++ [a]
```

```
ghci> preorder t
[5, 3, 8, 1, 4, 6]
ghci> inorder t
[8, 3, 1, 5, 4, 6]
ghci> postorder t
[8, 1, 3, 6, 4, 5]
```

```
      5
     / \
    3   4
   / \   \
  8   1   6
```

Ćwiczenie

Zdefiniuj typ reprezentujący drzewa o dowolnej liczbie poddrzew:

```
data Tree a = ...
```

Napisz funkcję obliczającą głębokość takiego drzewa:

```
depth :: Tree a -> Int
...
```

Typ konstruktora

```haskell
data Person = Person String String Int

ghci> :t Person
Person :: String -> String -> Int -> Person

ghci> :t Person "Jan"
Person "Jan" :: String -> Int -> Person

ghci> :t Person "Jan" "Kowalski"
Person "Jan" "Kowalski" :: Int -> Person

ghci> :t Person "Jan" "Kowalski" 22
Person "Jan" "Kowalski" 22 :: Person
```

Record syntax

```
data Person = Person String String Int

p = Person "Jan" "Kowalski" 22

ghci> let name (Person n _ _) = n
ghci> name p
"Jan"

data Person = Person { name :: String,
                       surname :: String,
                       age :: Int }

ghci> :t name
name :: Person -> String
ghci> name p
"Jan"
```