Programowanie funkcyjne

HASKELL

Typy podstawowe

- Bool – typ logiczny (True, False)
- Char – typ pojedynczego znaku ('a', 'A', '_',...)
- String – ciągi znaków ("Haskell", "Prolog", "",....)
- Int – (fixed-precision integers) liczby całkowite z przedziału $[-2^{31},2^{31}-1]$
- Integer – (arbitrary-precision integers) dowolne liczby całkowite
- Float – liczby zmiennoprzecinkowe pojedynczej precyzji
- Double – liczby zmiennoprzecinkowe podwójnej precyzji

Typy list

```
[False, True, True]          :: [Bool]
['a','b','c','d']            :: [Char]
["Curry", "Huskell"]         :: [String]
[['a','b'],['c','d']]        :: [[Char]]
```

Typy krotek

```
(False, True)                :: (Bool, Bool)
(False, 'a', True)           :: (Bool, Char, Bool)
("Haskell", False, 'a')      :: (String, Bool, Char)
```

Typy funkcji

```
f       :: T1 -> T2
isDigit :: Char -> Bool

add     :: (Int,Int) -> Int
add(x,y) = x+y

zeroton :: Int -> [Int]
zeroton = [0..n]
```

Curried function

```
add'  :: Int -> Int -> Int
```
oznacza
```
add'  :: Int -> (Int -> Int)
add' x y = x+y          (add' 1 :: Int -> Int)

mult   :: Int - > Int -> Int -> Int
```
oznacza:
```
mult   :: Int - > (Int -> (Int -> Int))
mult  x y z = x * y * z

mult x  y z   oznacza  ((mult x) y) z
```

Typy polimorficzne

```
Prelude> length [1,2,3,4,5]
5
Prelude> length ["Curry","Haskell"]
2
Prelude> length [fst, snd]
2
```

```
length  :: [a] -> Int

fst     :: (a,b) -> a
head    :: [a] -> a
take    :: Int -> [a] -> [a]
zip     :: [a] -> [b] -> [(a,b)]
```

Overloaded types

```
Prelude> 1+2
3
Prelude> 1.1+2.1
3.2
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

nazwa klasy — zmienna typowa

(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
abs :: Num a => a -> a
signum :: Num a => a -> a

type Name = String
type Age = Int

data People = Person Name Age
deriving (Show)

type Person People = (Name, Age)

Wartości tego typu:
Person "Anna Nowak" 23
Person "Jan" 30

("Anna Nowak",23)
("Jan", 30)

Podstawowe klasy
Klasa Eq

Eq - dla typów używanych w porównaniach (==,/=)
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool

Wszystkie typy podstawowe są instancjami klasy Eq, typy list i krotek.

```
Prelude> False==False
True
Prelude> 'a'=='b'
False
Prelude> "Haskell"=="Haskell"
True
Prelude> [1,2,3]==[1,2,3,4]
False
Prelude> ('a',False)==('a',False)
True
```

Podstawowe klasy
Klasa Ord

Ord - dla typów używanych w porządkowaniu wartości
(<),(<=),(>),(>=) :: a -> a -> Bool
min, max :: a -> a -> a

Wszystkie typy podstawowe są instancjami klasy Ord, typy list i krotek.

```
Prelude> False<True
True
Prelude> min 'a' 'b'
'a'
Prelude> "Curry" > "Haskell"
False
Prelude> [1,2] < [1,2,3]
True
Prelude> ('a',1) < ('b',0)
True
Prelude> ('a',1) < ('a',0)
False
```

Podstawowe klasy
Klasa Show

Show - klasa typów, których wartości mogą być konwertowane do ciągu znaków przy użyciu metody:

show :: a -> String

Wszystkie typy podstawowe są instancjami klasy Show, typy list i krotek.

```
Prelude> show True
"True"
Prelude> show 'a'
"'a'"
Prelude> show [1,2,3]
"[1,2,3]"
Prelude> show (1,'a',False)
"(1,'a',False)"
```

```
Prelude> let x=2
Prelude> let y=3
Prelude> "Suma " ++ show x ++ " i " ++ show y ++ " wynosi " ++ show (x+y) ++ "."
"Suma 2 i 3 wynosi 5."
```

Podstawowe klasy
Klasa Read

Read - klasa typów (dualna do Show), która zawiera typy, których wartości mogą być konwertowane z ciągu znaków przy użyciu metody:

read :: String -> a

Wszystkie typy podstawowe są instancjami klasy Read, typy list i krotek.

```
Prelude> read "False"
*** Exception: Prelude.read: no parse
Prelude> read "False"::Int
*** Exception: Prelude.read: no parse
Prelude> read "False"::Bool
False
```

```
Prelude> read "'a'":: Char
'a'
Prelude> read "100"::Int
100
Prelude> read "[1,2,3,4,5]"::[Int]
[1,2,3,4,5]
Prelude> read "('a',True,2)"::(Char,Bool,Int)
('a',True,2)
Prelude> read "12"+3
15
```

Podstawowe klasy
Klasa Num

Num - klasa typów, które są instancjami klasy Eq, Show, ale których wartości są liczbowe i do których można użyć metod:

(+), (-), (*) :: a -> a -> a

negate, abs, signum :: a -> a

Typy podstawowe Int, Integer, Float są instancjami klasy Num.

```
Prelude> 1+2
3
Prelude> 3.1*2
6.2
Prelude> negate 100
-100
Prelude> abs (-100)
100
Prelude> signum 100
1
```

Podstawowe klasy
Klasa Integral

Integral - klasa typów, które są instancjami klasy Num, ale których wartości są całkowite i do których używa się metod:

div :: a -> a -> a

mod :: a -> a -> a

Typy podstawowe Int, Integer są instancjami klasy Integral.

```
Prelude> div 9 2
4
Prelude> 10 `div` 4
2
Prelude> 10 `mod` 6
4
```

Podstawowe klasy
Klasa Fractional

Fractional - klasa typów, które są instancjami klasy Num, ale których wartości nie są całkowite i do których używa się metod:

(/) :: a -> a -> a

recip :: a ->

Typ podstawowy Float jest instancją klasy Fractional.

```
Prelude> 9/4
2.25
Prelude> 7.5/3.4
2.2058823529411766
Prelude> recip 2.0
0.5
Prelude> recip 10
0.1
```

Klasy typów

Definicja (deklaracja) klasy

```
class Nazwa-klasy zmienne-typowe where
  nazwa-funkcji/operatora :: typ-funkcji/operatora
  definicja-niektórych-funkcji/operatorów
```

Aby typ danych stał się egzemplarzem klasy, należy użyć konstrukcji:

```
instance Nazwa-klasy Nazwa-typu where
  przeciążenie-wymaganych-funkcji/operatorów
```

lub

```
data definicja-typu deriving (lista-klas)
```

(Klauzula deriving użyta do tworzenia instancji klas)

Definiowanie klas typów - przykłady

class Eq a **where**

(==),(/=) :: a -> a -> Bool

x /=y = not (x==y)

Typ a jest instancją klasy Eq, jeżeli istnieją dla niego operacje == i /=

```
Prelude> :type (==)
(==) :: Eq a => a -> a -> Bool
Prelude> :type (/=)
(/=) :: Eq a => a -> a -> Bool
Prelude> :type elem
elem :: Eq a => a -> [a] -> Bool
```

Jeżeli typ a jest instancją Eq, to (==) ma typ a -> a -> Bool

Jeżeli typ a jest instancją Eq, to elem ma typ a -> [a] -> Bool

Klasa Enum

class Enum a **where**

succ, pred :: a -> a

toEnum :: Int -> a

fromEnum :: a -> Int

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

```
*Main> succ Mon
Tue
*Main> pred Mon
*** Exception: pred{Day}: tried to take `pred' of first tag in enumeration
*Main> fromEnum Fri
4
```

Dziedziczenie

```
class Eq a => Ord a where
        (<),(<=),(>),(>=) :: a -> a -> Bool
        min, max          :: a -> a -> a
        min x y | x<=y        = x
                | otherwise = y
        max x y | x<=y        = y
                | otherwise = x
```

Ord jest podklasą Eq (każdy typ klasy Ord musi być też instancją klasy Eq)

- Uwaga: Dziedziczenie może być wielokrotne

Dziedziczenie

```
class (Eq a, Show a) => Num a where
        (+), (-), (*)   :: a -> a -> a
        negate          :: a -> a
        abs, signum :: a -> a
        x - y           =  x + negate y
        negate x        = 0 - x
```

```
Prelude> abs (-5)
5
Prelude> negate 8
-8
Prelude> signum (-2)
-1
```

Definiowanie klas typów - przykłady

```
instance Ord Bool where
        False < True  =  True
        _ < _         =  False
        b <= c        = (b < c) && (b ==c)
        b > c         = c < b
        b >= c        = c <= b
```

Deklarowanie instancji klas typów

```
data Bool = False | True
          deriving(Eq, Ord, Show, Read)
```

```
instance Eq Bool where
    False == False  = True
    True == True    = True
    _ == _          = False
```

Bool jest instancją Eq i definicja operacji (==) jest j.w. (metoda)

Instancje typów

```
class Eq a where
    (==),(/=) :: a -> a -> Bool
```

```
data Color = Red
           | Orange
           | Yellow
           | Green
           | Blue
           | Purple
           | White
           | Black
           | Custom Int Int Int  -- R G B components, respectively

instance Eq Color where
    Red == Red = True
    Orange == Orange = True
    Yellow == Yellow = True
    Green == Green = True
    Blue == Blue = True
    Purple == Purple = True
    White == White = True
    Black == Black = True
    (Custom r g b) == (Custom r' g' b') =
        r == r' && g == g' && b == b'
    _ == _ = False
```

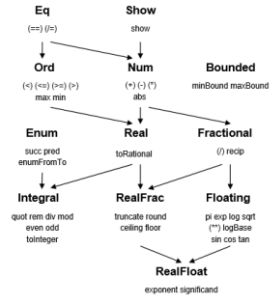
```
data Tree a = Empty | Node a (Tree a) (Tree a)
            deriving(Show)
```

```
*Main>  elem Empty [(Node 1 Empty Empty), Empty]

<interactive>:11:2:
    No instance for (Eq (Tree a0)) arising from a use of 'elem'
```

```
instance Eq a => Eq (Tree a) where
    Empty == Empty                      = True
    (Node a1 l1 r1) == (Node a2 l2 r2) = (a1 == a2) && (l1 == l2) && (r1 == r2)
    _ == _                              = False
```

```
*Main> elem Empty [(Node 1 Empty Empty), Empty]
True
```




```
qsort :: [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs)
               ++ [x]
               ++ qsort (filter (>= x) xs)
```

```
Prelude> :load "qsort.hs"
[1 of 1] Compiling Main             ( qsort.hs, interpreted )

qsort.hs:3:31:
    No instance for (Ord a) arising from a use of '<'
    Possible fix:
      add (Ord a) to the context of
        the type signature for qsort :: [a] -> [a]
    In the first argument of 'filter', namely '(< x)'
    In the first argument of 'qsort', namely '(filter (< x) xs)'
    In the first argument of '(++)', namely 'qsort (filter (< x) xs)'
Failed, modules loaded: none.
Prelude>
```

qsort :: Ord a => [a] -> [a]

```
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs)
               ++ [x]
               ++ qsort (filter (>= x) xs)
```

```
*Main> :load "qsort.hs"
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> qsort [2,4,3,5,61,0,9,-3]
[-3,0,2,3,4,5,9,61]
*Main> qsort ['w','y','a','k','b']
"abkwy"
```

Enkapsulacja

Struktura modułu

```
module Nazwa (...) where
...ciało-modułu...
```

- Nazwa modułu musi być napisana z dużej litery i taką samą nazwę powinniśmy nadać plikowi z rozszerzeniem .hs, w którym moduł zapisujemy.
- Nawiasem (...) obejmuje się listę nazw funkcji i typów danych, z których użytkownik może korzystać. Można też tę część nagłówka modułu pominąć, wówczas wszystkie funkcje i typy danych będą dostępne.
- W skład ciała modułu wchodzą definicje klas typów, typów danych i funkcji.

Enkapsulacja

Załóżmy, że w module o nazwie M zdefiniowano funkcje **f** i **g**,

typy danych **A** z konstruktorami **Ka1**, **Ka2**, **Ka3** oraz

typ danych **B** z konstruktorami **Kb1**, **Kb2**, **Kb3**.

Jeśli na zewnątrz mają być widoczne: funkcja f, typ A ze wszystkimi konstruktorami, typ B z konstruktorami Kb1, Kb3, to początek pliku z modułem powinien wyglądać następująco:

```
module M (A(..), B(Kb1,Kb3), f) where
...
```

Z modułu korzystamy w innych modułach po ich zaimportowaniu:

```
import Nazwa
```

Literatura

- B.O'Sullivan,J.Goerzen, D.Stewart, Real World Haskell, O'REILLY, 2008.
- K.Doets, J.van Eijck, The Haskell Road to Logic, Math and programming, 2004.
- G.Brzykcy, A.Meissner, Programowanie w Prologu i programowanie funkcyjne, Wyd.PP, 1999.
- Miran Lipovaca, Learn You a Haskell for Great Good!