Programowanie funkcyjne

HASKELL

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
```

Klasy typów

Definicja 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

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

Deklarowanie instancji klas typów

data Bool = False | True

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

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

Dziedziczenie

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
```

Dziedziczenie

class Eq a => Ord a where
(<), (<=), (>=), (>) :: a -> a -> Bool
max, min :: a -> a -> a
x < y = x <= y && x/= y

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

- Uwaga: Dziedziczenie może być wielokrotne

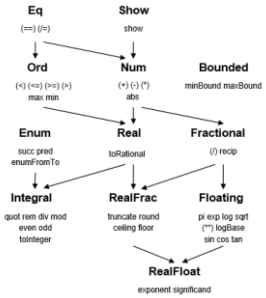
data Tree a = Empty | Node a (Tree a) (Tree a)
deriving Show

```
*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
```



Wbudowane klasy i ich typy

Typ	Za pomocą instance	Za pomocą deriving
Bool		Eq Ord Enum Bounded
Int	Integral Bounded	
Integer	Integral	
Float	RealFloat	
Double	RealFloat	
Char	Eq Ord Enum	
[a]		Eq Ord
(a,b)		Eq Ord Bounded

Podstawowe klasy typów (Prelude.hs)

Eq, Ord, Show, Read, Num, Enum

```
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"
```

Klasa Show

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

```
Prelude> show 12345
"12345"
```

```
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."
```

Klasa Show

```
show :: Show a => a -> String
showsPrec :: Show a => Int -> a -> String -> String
showList :: Show a => [a] -> String -> String
```

```
instance Show Color where
    show Red = "Red"
    show Orange = "Orange"
    show Yellow = "Yellow"
    show Green = "Green"
    show Blue = "Blue"
    show Purple = "Purple"
    show White = "White"
    show Black = "Black"
    show (Custom r g b) =
        "Custom " ++ show r ++ " " ++ show g ++ " " ++ show b
```

Klasa Read

```
read :: Read a => String -> a
```

```
*Main> (read "12") :: Float
12.0
*Main> read "12"+3
15
```

Klasa Num

```
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
```

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
           deriving (Show, Enum)
*Main> succ Mon
 Tue
*Main> pred Mon
    *** Exception: tried to take 'pred' of first tag in enumeration
*Main> (toEnum 5)  Day
 Sat
 *Main> fromEnum Mon
  0
```

Unie w C

Unia jest specjalnym rodzajem struktury, w której „aktywne” jest tylko jedno pole.
Unię deklaruje się w podobny sposób jak strukturę, np.:

```
union nazwa
{
    char c;
    int i;
    double f;
} x;
```

```
x.c = 'a';
x.d = 12.15;
```

Interpretacja graficzna unii.

Bezpieczne unie w Haskellu

Przykład unii z dwoma elementami:

```
data Unia typ1 typ2 = Jedno typ1 | Drugie typ2
        deriving(Eq, Show)
```

Unie w Haskellu z przykładowymi danymi i funkcjami (następna strona)

```
data Unia typ1 typ2 = Jedno typ1 | Drugie typ2
    deriving(Eq, Show)

--funkcja testowa
wybierzJedno [ ] = [ ]
wybierzJedno (( Jedno x) : o) = x : wybierzJedno o
wybierzJedno (( Drugie x) : o) = wybierzJedno o

-- dane testowe
listaZnakowILiczb = [ Jedno 'k', Drugie 13, Jedno 'o', Drugie 100, Drugie 5]

unijnaListaLiczb = map Drugie [1, 2, 3]
unijnaListaZnakow = map Jedno "ala"

razem = listaZnakowILiczb
      ++ unijnaListaLiczb
      ++ unijnaListaZnakow

zwierz = wybierzJedno razem
```

```
*Main> :t Jedno
Jedno :: typ1 -> Unia typ1 typ2
*Main> :t Drugie
Drugie :: typ2 -> Unia typ1 typ2
*Main> :t wybierzJedno
wybierzJedno :: [Unia t t1] -> [t]
*Main> :t listaZnakowILiczb
listaZnakowILiczb :: [Unia Char Integer]
*Main> :t unijnaListaLiczb
unijnaListaLiczb :: [Unia typ1 Integer]
*Main> :t unijnaListaZnakow
unijnaListaZnakow :: [Unia Char typ2]
*Main> :t razem
razem :: [Unia Char Integer]
```

```
*Main> :t razem
razem :: [Unia Char Integer]
*Main> razem
[Jedno 'k',Drugie 13,Jedno 'o',Drugie 100,Drugie 5,Drugie 1,
Drugie 2,Drugie 3,Jedno 'a',Jedno 'l',Jedno 'a']
*Main> :t zwierz
zwierz :: [Char]
*Main> zwierz
"koala"
```

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!