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)
```

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 => 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'
```

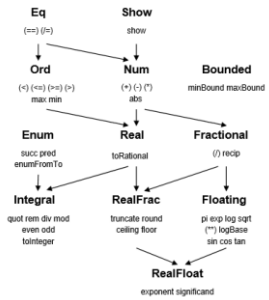
instance Eq a => Eq (Tree a) where

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 Read

read :: Read a => String -> a

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

Klasa Enum

```
class Enum a where
  succ, pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
```

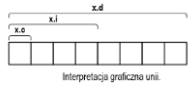
```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
          deriving (Show, Enum)
*Main> succ Mon
 Tue
*Main> pred Mon
    *** Exception: tried to take 'pred' of first tag in enumeration
*Main> (toEnum 5)  Day
 Sat
 *Maini> fromEnum Mon
  0
```

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"
```

Monady w Haskellu

Monady wykorzystywane są w Haskellu.

Struktura monady nadaje się do specyfikacji:

a) operacji wejścia/wyjścia,

b) wyłapywania wyjątków (np. takich jak dzielenie przez zero),

c) interfejsów graficznych.

W ujęciu Haskellowym Monadę tworzy konstruktor typów **m**, wraz z pewnymi szczególnymi operacjami wchodzącymi w skład klasy Monad.

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
```

Interpretacja konstruktora m jest następująca:

jeśli a jest typem wartości, m a reprezentuje ***typ obliczeń*** zwracających wartość typu a.

Obliczenie" i „zwracanie wartości" należy rozumieć abstrakcyjnie. Obliczenie typu m a może być na przykład:

- po prostu wartością typu a - m oznacza wtedy trywialne obliczenie;
- wartością typu a lub wartością wyjątkową, reprezentującą błędne obliczenie;
- zbiorem możliwych wartości typu a - m a oznacza wtedy obliczenie niedeterministyczne;
- programem z efektami ubocznymi, reprezentowanym przez funkcję typu s -> (a, s), gdzie s jest typem stanu modyfikowanego przez funkcję.

Operacja **return** konstruuje obliczenie zwracające daną wartość.

(f >>= g) to sekwencyjne złożenie obliczeń f i (g a), gdzie a jest wartością obliczenia f.

(f >> h) to sekwencyjne złożenie obliczeń f i h, przy czym h nie zależy od wartości obliczenia f.

>> można zdefiniować przy pomocy >>= (ćwiczenie)

Programy interaktywne

Typ reprezentujący operacje IO

- funkcja zmieniająca „stan świata"
  type IO = World -> World
- funkcja zmieniająca „stan świata" i zwracająca wynik
  type IO a = World -> (a, World)

Akcje

Akcja to wyrażenie typu IO a

IO Char (typ akcji zwracającej znak)

IO () (typ akcji zwracającej pustą krotkę)

Typ jednostkowy

data () = ()

Podstawowe akcje

- akcja **getChar** wczytuje znak z klawiatury, wyświetla go na ekranie i zwraca jako rezultat

  getChar :: IO Char

- akcja **putChar c** wyświetla znak c na ekranie i zwraca pustą krotkę

  putChar :: Char -> IO ()

- akcja **return** v zwraca wartość v bez jakichkolwiek interakcji

  return :: a -> IO a

  return v = \world -> (v, world)

Operator sekwencji

(>>=) :: IO a -> (a -> IO b) -> IO b

f >>= g = \world -> case f world of

(v, world') -> g v world'

Uwaga

Jak w przypadku parserów zamiast operatora >>= można korzystać z notacji **do**

Przykład

```
a :: IO (Char, Char)
a = do x <- getChar
       getChar
       y <- getChar
       return (x, y)
```

```
*Main> a
123456
('1','3')
*Main> *Main>
('4','6')
```

Sekwencję elementów monady tłumaczy się na notację (>>=) i (**>>**) następująco:

```
do a <- e        |-->    e >>= \ a -> do e'
   e'

do e             |-->    e >> do e'
   e'

do let x = e     |-->    let x = e
   e'                     in  do e'

do e             |-->    e
```

```
echo :: IO ()
echo = do line <- getLine
          putStr line

palindrom :: IO ()
palindrom = do putStr "Napisz cos: "
               line <- getLine
               let line' = filter (not . (==) ' ') (map toLower line)
               if line' == reverse line'
                   then putStr ("'" ++ line ++ "' jest palindromem!\n")
                   else putStr ("'" ++ line ++ "' nie jest palindromem.\n")
```

```
*Main> palindrom
Napisz cos: einawargorp
'einawargorp' nie jest palindromem.
*Main> palindrom
Napisz cos: abcdcba
'abcdcba' jest palindromem!
```

getLine

```
getLine :: IO String
getLine = do  x <- getChar
              if x == '\n'  then return []
                else
                  do xs <- getLine
                     return (x:xs)
```

putStr

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

putStrLn

```
putStrLn :: String -> IO ()
putStrLn xs = do
                putStr xs
                putChar '\n'
```

Przykład

```
strlen :: IO ()
strlen = do putStr "Enter a string: "
            xs <- getLine
            putStr "String ma "
            putStr (show (length xs))
            putStrLn " znakow"
```

```
*Main> strlen
Enter a string: Programowanie funkcyjne
String ma 23 znakow
*Main> strlen
Enter a string: Haskell
String ma 7 znakow
```

Przykład

```
power = do putStr "Podaj liczbe: "
           n <- getLine
           let x = read n
               y = x^2
           putStrLn (n ++ " do kwadratu: " ++ show y)
```

```
*Main> power
Podaj liczbe: 15
15 do kwadratu: 225
```

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!