# Programowanie funkcyjne
(wykład 5.)


Roman Dębski

Instytut Informatyki, AGH


30 listopada 2021



## Plan wykładu


1. Operacje wejścia/wyjścia
2. Wzorce: funktor, funktor aplikatywny, monoid
3. Dodatki [materiał opcjonalny]): Foldable, Traversable




## Plan wykładu


1. Operacje wejścia/wyjścia
2. Wzorce: funktor, funktor aplikatywny, monoid
3. Dodatki [materiał opcjonalny]): Foldable, Traversable




## Problem z I/O w Haskellu

*Problem*: program w Haskellu to złożona funkcja (matematyczna/"czysta", czyli bez ***skutków ubocznych***). Jak zatem przy tym założeniu realizować **operacje I/O**?

*Proponowane rozwiązanie*: rozdzielnie przetwarzania "czystego" od "nieczystego" :) i reprezentacja *skutków ubocznych* jako **wartości** typu **IO a**.

*"A value of type IO a is an action' that, **'when performed, may do some input/output, before delivering a value of type a**. You will often also find them called computations".*

— Simon Peyton Jones

*"Actions are defined rather than invoked within the expression language of Haskell. Evaluating the definition of an action doesn't actually cause the action to happen. Rather, the invocation of actions takes place outside of the expression evaluation [...]".*

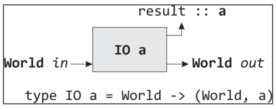
— https://www.haskell.org/tutorial/io.html

*"**Laziness and side effects are, from a practical point of view, incompatible**. If you want to use a lazy language, it pretty much has to be a purely functional language; if you want to use side effects, you had better use a strict language".*

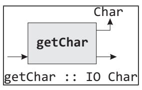
— Simon Peyton Jones

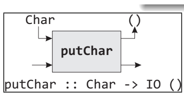


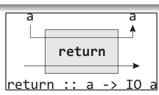
## Wybrane operacje I/O: reprezentacja graficzna




```haskell
-- pseudo-code, it does not compile!
main :: World -> (World, ())
main world0 =
  let (world1, a) = getChar world0
      (world2, b) = getChar world1
  in (world2, ())
```



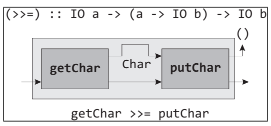




```
ghci> getChar
```

```
ghci> putChar 'a'
a
```

```
ghci> let retA = return 'a'
ghci> retA
'a'
```




```
ghci> let echo = getChar >>= putChar
ghci> echo
mm
```



## I/O: *do notation* [ *a convenient way to define a sequence of actions* ]

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

```haskell
-- (>>) :: IO a -> IO b -> IO b
main = putStr "Hello" >>
       putStrLn " World"
```

```haskell
main = do
  putStr "Hello"
  putStrLn " World"
```

```haskell
-- (>>=) :: IO a -> (a -> IO b) -> IO b
main = putStrLn "Your name?" >>
       getLine >>=
       \n -> putStrLn ("Hello, " ++ n)
```

```haskell
main = do
  putStrLn "Your name?"
  n <- getLine
  putStrLn ("Hello, " ++ n)
```

```haskell
main = do
  a <- return "a"
  b <- return "b"
  return () -- compare with return in C!
  return 1  -- !
  putStrLn $ a ++ " " ++ b -- !
```

```haskell
main = do
  let a = "a"
      b = "b"
  return ()
  return 1
  putStrLn $ a ++ " " ++ b
```



## I/O: rekurencyjne definicje funkcji [ dwa przykłady ]

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

main = do
  line <- getLine'
  putStrLn line
```

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

putStrLn na podstawie putStr

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



## Praca z plikami [przykład]

```haskell
import System.Environment
import System.IO
import Data.Char(toUpper)

-- openFile :: FilePath -> IOMode -> IO Handle
main = do
  (inFileName:outFileName:_) <- getArgs
  inHdlr <- openFile inFileName ReadMode
  outHdlr <- openFile outFileName WriteMode
  inpStr <- hGetContents inHdlr
  hPutStr outHdlr (map toUpper inpStr)
  hClose inHdlr
  hClose outHdlr
```

```haskell
main = do
  (inFileName:outFileName:_) <- getArgs
  inpStr <- readFile inFileName
  writeFile outFileName (map toUpper inpStr)
```

*uwaga*: lazy I/O

## I/O i obsługa błędów [przykład]

```haskell
import System.Environment
import System.IO
import System.IO.Error
import Control.Exception

main = do (fileName:_) <- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has " ++
                     show (length (lines contents)) ++ " lines!"
       `catch` (\err -> if isDoesNotExistError err
                        then putStrLn "The file doesn't exist!"
                        else ioError err)
```

```haskell
ghci> :t catch
catch :: Exception e => IO a -> (e -> IO a) -> IO a
ghci> :t isDoesNotExistError
isDoesNotExistError :: IOError -> Bool
```

*uwaga*: obsługa błędów w kodzie I/O vs. poza I/O (np. Maybe, Either)



## Plan wykładu

1. Operacje wejścia/wyjścia
2. Wzorce: funktor, funktor aplikatywny, monoid
3. Dodatki [materiał opcjonalny]: Foldable, Traversable



## Funktor [ *Mappable* ] : definicja i przykłady (*uwaga*: uzupełnienie podczas 7. wykładu)

*a type that can be mapped over* vs. ... vs. *homomorphism between categories* :)

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) ->   f a   ->  f b
--fmap :: (a -> b) -> Maybe a -> Maybe b
```

*Functor laws*

```haskell
fmap id = id
fmap (g . f) = fmap g . fmap f
```

```haskell
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

```haskell
ghci> fmap (^3) (Just 4)
Just 64
```

```haskell
instance Functor [] where
  fmap = map
```

```haskell
ghci> fmap show [1..5]
["1","2","3","4","5"]
```

```haskell
instance Functor IO where
  fmap f action = do
    result <- action
    return (f result)
```

```haskell
ghci> fmap reverse getLine
rats
"star"
```



## Funktor: rozszerzenie *DeriveFunctor**

```haskell
data BinTree a = EmptyBT
               | NodeBT a (BinTree a) (BinTree a)
               deriving (Show)

instance Functor BinTree where
  fmap g EmptyBT          = EmptyBT
  fmap g (NodeBT x lt rt) = NodeBT (g x) (fmap g lt) (fmap g rt)
```

```haskell
ghci> let t1 = NodeBT 1 (NodeBT 4 EmptyBT EmptyBT) EmptyBT
ghci> fmap (*2) t1
NodeBT 2 (NodeBT 8 EmptyBT EmptyBT) EmptyBT
```

```haskell
{-# LANGUAGE DeriveFunctor #-}
data BinTree a = EmptyBT
               | NodeBT a (BinTree a) (BinTree a)
               deriving (Show, Functor)
```

**A given type has at most one valid instance of Functor; it can be automatically derived by GHC for many data types*



## Funktor aplikatywny: definicja

*A functor that supports function application within its context; it encapsulates certain sorts of 'effectful' computations in a functionally pure way, and encourages an 'applicative' programming style*

```haskell
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
-- fmap ::   (a -> b) -> f a -> f b
```

*Applicative functor laws*

```haskell
pure id <*> v = v                            -- Identity
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w -- Composition
```

```haskell
fmap g x = pure g <*> x = g <$> x
```

*Przykładowe zastosowania?*



## Funktory aplikatywne: przykłady

```haskell
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> w = fmap f w
```

```haskell
(++) <$> Just "me and "<*>pure "Haskell"
Just "me and Haskell"
```

```haskell
instance Applicative IO where
  -- returnIO :: a -> IO a
  pure = returnIO
  a <*> b = do
    f <- a
    x <- b
    return (f x)
```

```haskell
myAction = (++) <$> getLine <*> getLine
ghci> myAction
abra
kadabra
"abrakadabra"
```

```haskell
instance Applicative [] where
  pure x = [x]
  fs <*> xs =
    [f x | f <- fs, x <- xs]
```

```haskell
import Control.Applicative
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs =
   ZipList (zipWith (\f x -> f x) fs xs)
```



## Monoid [ *Appendable, Concatable* ]: definicja i przykłady

*A type together with an associative binary operation* (in Haskell -* ***mappend****) which has an identity element (in Haskell -* ***mempty****); i.e. it's a semigroup with identity :)*

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  {-# MINIMAL mempty, mappend #-}
```

*Monoid laws*

```haskell
mempty `mappend` x = x
x `mappend` mempty = x
(x `mappend` y) `mappend` z =
   x `mappend` (y `mappend` z)
```

```haskell
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

```haskell
-- import Data.Monoid
ghci> [1,2] `mappend` [3,4]
[1,2,3,4]
```

```haskell
instance Monoid b => Monoid (a->b) where
  mempty _ = mempty
  mappend f g x = f x `mappend` g x
```

```haskell
-- import Data.Monoid
ghci> mappend (*2) (+10) (Sum 2)
Sum {getSum = 16}
```

* łączność → możliwość zrównoleglenia obliczeń, 'agregowalność' → deklaratywny fold (automatycznie generowany)



## Plan wykładu

1. Operacje wejścia/wyjścia
2. Wzorce: funktor, funktor aplikatywny, monoid
3. Dodatki [materiał opcjonalny]: Foldable, Traversable

## Foldable

```haskell
class Foldable (t :: * -> *) where
  fold    :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr   :: (a -> b -> b) -> b -> t a -> b
  foldl   :: (a -> b -> a) -> a -> t b -> a
  foldr1  :: (a -> a -> a) -> t a -> a
  foldl1  :: (a -> a -> a) -> t a -> a
  {-# MINIMAL foldMap | foldr #-}
```

```haskell
instance Foldable [] where
  foldMap g = mconcat . map g
```

```haskell
{-# LANGUAGE DeriveFoldable #-}
data BinTree a = EmptyBT
               | NodeBT a (BinTree a) (BinTree a)
               deriving (Show, Foldable)
```



## Traversable [traversable functor]

```haskell
class (Functor t, Foldable t) => Traversable (t :: * -> *) where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
  {-# MINIMAL traverse | sequenceA #-}
```

```haskell
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}
data BinTree a = EmptyBT
               | NodeBT a (BinTree a) (BinTree a)
               deriving (Show, Functor, Foldable, Traversable)
```

*"Where Foldable gives you the ability to go through the structure processing the elements (foldr) but throwing away the shape, Traversable allows you to do that whilst preserving the shape and, e.g., putting new values in".*

— https://wiki.haskell.org/Foldable_and_Traversable



## Bibliografia


- Simon Thompson, Haskell: The Craft of Functional Programming, Addison-Wesley Professional, 2011
- Graham Hutton, Programming in Haskell, Cambridge University Press, 2007
- https://wiki.haskell.org
- http://learnyouahaskell.com
- http://book.realworldhaskell.org/read
- https://wiki.haskell.org/Typeclassopedia
- https://research.microsoft.com/en-us/um/people/simonpj/papers/stm/beautiful.pdf
- https://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/mark.pdf