Монади

Монади – це засіб вказати послідовність виконання дій.

Клас монад

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

  p >> q = p >>= \ _ -> q
  fail s = error s
```

Клас монад

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b      -- bind
   (>>)   :: m a -> m b -> m b             -- then
    fail  :: String -> m a

  p >> q = p >>= \ _ -> q
  fail s = error s
```

Зв’язування (Bind)
Послідовне компонування
(Sequentially composition)

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

Зв’язування (Bind)
Послідовне компонування
(Sequentially composition)

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

```
Prelude> getLine >>= \text -> putStrLn $ "Ви
   сказали '" ++ text ++ "'"
```

Зв’язування (Bind)
Послідовне компонування
(Sequentially composition)

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

```
Prelude> getLine >>= \text -> putStrLn $ "Ви
   сказали '" ++ text ++ "'"
Hello!
Ви сказали 'Hello!'
```

Потім (Then)

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

Потім (Then)

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

```
Prelude> putStrLn "Спочатку це" >>
   putStrLn "Потім це"
```

Потім (Then)

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

```
Prelude> putStrLn "Спочатку це" >>
   putStrLn "Тепер це"
```

```
Спочатку це
Тепер це
```

Щоб тип був монадою:

він має бути втіленням класу Monad

та

програміст гарантує виконання аксіом

1. return x>>=f ≡ f x
2. f >>= return ≡ f
3. f >>= (\x -> g x >>= h) ≡ (f>>=g)>>=h

3’. f >>= (g >>= h) ≡ (f>>=g)>>=h

do-нотація

do { e } перетворюється на e

do { e1; e2 } перетворюється на e1 >> e2

do {x<-e1; e2} перетворюється на

e1 >>= \x -> e2

do {let decls; es} перетворюється на

let decls in do {es}

do-нотація

```
Prelude>getLine >>= \text -> putStrLn $ "Ви
   сказали '" ++ text ++ "'"
```

```
Prelude> do text<-getLine; putStrLn $ "Ви
   сказали '" ++ text ++ "'"
```

do-нотація

```
Prelude>getLine >>= \text -> putStrLn $ "Ви сказали '" ++ text ++ "'"
```

```
Prelude> do text<-getLine; putStrLn $ "Ви сказали '" ++ text ++ "'"
ТАК
Ви сказали 'ТАК'
```

do-нотація

```
Prelude>putStrLn  "Ви сказали "  >>
   putStrLn  "Ще сказали"
```

```
Prelude> do putStrLn  "Ви сказали ";
   putStrLn  "Ще сказали"
Ви сказали
Ще сказали
```

Приклади монад

тип m a – “значення типу a, отримане певним чином”

- [ ] “можливо кілька значень”
- Maybe “одне або жодного значення”
- Either “одне або спеціальне значення”
- IO “з побічними ефектами”
- Writer
- Reader
- State
- ...

```
sequence :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

```
sequence :: Monad m => [m a] -> m [a]

> sequence [Just 10, Just 20, Just 30]
Just [10,20,30]

> sequence [Just "ab", Just "fg", Just "kl"]
Just ["ab","fg","kl"]

> sequence [putStrLn "десять", putStrLn "двадцять", putStrLn
   "тридцять"]
десять
двадцять
тридцять
[(),(),()]
```

```
sequence_ :: Monad m => [m a] -> m ()

> sequence_ [Just 10, Just 20, Just 30]
Just ()

> sequence_ [Just "ab", Just "fg", Just "kl"]
Just ()

Prelude> sequence_ [putStrLn "десять", putStrLn
   "двадцять", putStrLn "тридцять"]
десять
двадцять
тридцять
```

mapM :: Monad m => (a -> m b) -> [a] -> m [b]

sequence . map f

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

sequence_ . map f

mapM :: Monad m => (a -> m b) -> [a] -> m [b]

sequence . map f

```
> mapM print [10,20,30]
10
20
30
[(),(),()]
```

mapM :: Monad m => (a -> m b) -> [a] -> m [b]

sequence . map f

```
Prelude> mapM print ["as", "fg", "rt"]
"as"
"fg"
"rt"
[(),(),()]
```

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

sequence_ . map f

> mapM_ print [10,20,30]

10

20

30

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

sequence_ . map f

> *mapM_ print ["as", "fg", "rt"]*

"as"

"fg"

"rt"

Монада IO

див. також System.IO

getChar :: IO Char
getLine :: IO String
getContents :: IO String
interact :: (String -> String) -> IO ()

putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
print :: Show a => a -> IO ()

```
data IOMode = ReadMode
            | WriteMode
            | AppendMode
            | ReadWriteMode

openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
hIsEOF :: Handle -> IO Bool
```

```
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar :: IO Char
getLine :: IO String
getContents :: IO String
```

```
hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()

putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
```

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()

bracket :: IO a ->(a -> IO b) -> (a -> IO c) -> IO c
bracket before after m = do
        x  <- before
        rs <- try (m x)
        _ <- after x
        case rs of
           Right r -> return r
           Left  e -> ioError e
```

```
bracket
    (openFile "filename" ReadMode)
    (hClose)
    (\fileHandle -> do { ... })
```

```
module Main   where

import System.IO
import Control.Exception

main = do hSetBuffering stdin LineBuffering
                  doLoop
doLoop = do putStrLn "Enter a command rFN wFN or q to quit:"
            command <- getLine
          case command of
            'q':_        -> return ()
            'r':filename -> do putStrLn ("Reading " ++ filename)
                               doRead filename
                               doLoop
            'w':filename -> do putStrLn ("Writing " ++ filename)
                               doWrite filename
                               doLoop
            _            -> doLoop
doRead filename = bracket (openFile filename ReadMode)
                          hClose
                  (\h -> do contents <- hGetContents h
                            putStrLn "The first 100 chars:"
                            putStrLn (take 100 contents))

doWrite filename = do putStrLn "Enter text to go into the file:"
                      contents <- getLine
                    bracket (openFile filename WriteMode) hClose
                            (\h -> hPutStrLn h contents)
```

```
module Main
  where

import System.IO
import Control.Exception

main = do hSetBuffering stdin LineBuffering
          doLoop
```

```
doLoop = do putStrLn "Enter a command rFN, wFN or q :"
            command <- getLine
            case command of
              'q':_        -> return ()
              'r':filename -> do putStrLn ("Reading " ++ filename)
                                 doRead filename
                                 doLoop
              'w':filename -> do putStrLn ("Writing " ++ filename)
                                 doWrite filename
                                 doLoop
              _            -> doLoop
```

```
doRead filename = bracket (openFile filename ReadMode)
                          hClose
                          (\h -> do contents <- hGetContents h
                                    putStrLn "The first 100 chars:"
                                    putStrLn (take 100 contents))

doWrite filename = do putStrLn "Enter text to go into the file:"
                      contents <- getLine
                      bracket (openFile filename WriteMode) hClose
                              (\h -> hPutStrLn h contents)
```

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

main = do
        [f1,f2] <- getArgs
        s <- readFile f1
        writeFile f2 (map toUpper s)
```

компіляція:

ghc file_2.hs -o file_2

виконання:

file_2.exe 1.txt 1_out.txt

1.txt

```
Loading packages
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Ok, modules loaded: Main.
120
Leaving GHCi.
```

1_out.txt

LOADING PACKAGES

LOADING PACKAGE GHC-PRIM ... LINKING ... DONE.

LOADING PACKAGE INTEGER-GMP ... LINKING ... DONE.

LOADING PACKAGE BASE ... LINKING ... DONE.

LOADING PACKAGE FFI-1.0 ... LINKING ... DONE.

OK, MODULES LOADED: MAIN.

120

LEAVING GHCI.