Функційне програмування мовою Haskell

Обчислення у контекстах

Класи:

- Functor
- Applicative (аплікативний функтор)
- Monad

1. Контейнерні типи даних
2. Обчислювальні контексти

Список, конструктор типу []

```
*Main> :k []
[] :: * -> *

*Main> :t []
[] :: [a]

Int
[Int]
(+) :: Int -> Int -> Int
```

Список, конструктор типу []

```
*Main> :k []
[] :: * -> *

*Main> :t []
[] :: [a]

Int
[Int]
(+) :: Int -> Int -> Int
> 1 + 2
> [1] + [2]
```

-

- []

data Maybe a

- []

data Maybe a

*Main> :k Maybe
Maybe :: * -> *

- []

data Maybe a

```
*Main> :k Maybe
Maybe :: * -> *

*Main> :t Just "abc"
Just "abc" :: Maybe [Char]

*Main> :t Nothing
Nothing :: Maybe a
```

- []

data Maybe a

*Main> :k Maybe
Maybe :: * -> *

*Main> :t Just "abc"
Just "abc" :: Maybe [Char]

*Main> :t Nothing
Nothing :: Maybe a

Int
Maybe Int
(+) :: Int -> Int -> Int

> 1 + 2
> (+) (Just 1) (Just 2)

- []
- Maybe

```
*Main> :k IO
IO :: * -> *
```

- []
- Maybe
- IO

- []
- Maybe
- IO

```
data  Either a b  =  Left a
                  | Right b
                  deriving (Eq, Ord, Read, Show)
```

- []
- Maybe
- IO

```
data  Either a b  =  Left a
                  | Right b
                  deriving (Eq, Ord, Read, Show)
```

```
*Main> :k Either
Either :: * -> * -> *
```

- []
- Maybe
- IO

```
data  Either a b  =  Left a
                  | Right b
                  deriving (Eq, Ord, Read, Show)
```

```
*Main> :k Either
Either :: * -> * -> *
```

Конструктор з двома параметрами. Треба з одним

- []
- Maybe
- IO

```
data  Either a b  =  Left a
                  | Right b
                  deriving (Eq, Ord, Read, Show)
```

```
*Main> :k Either
Either :: * -> * -> *
```

Конструктор з двома параметрами. Треба з одним

У конструктора (Either a) - тільки один параметр

- []
- Maybe
- IO

```
data  Either a b  =  Left a
                  | Right b
                  deriving (Eq, Ord, Read, Show)
```

```
*Main> :k Either
Either :: * -> * -> *
```

Конструктор з двома параметрами. Треба з одним

У конструктора (Either a) - тільки один параметр

```
*Main> :k Either String
Either String :: * -> *
```

- []
- Maybe
- IO

```
data  Either a b  =  Left a
                  | Right b
                  deriving (Eq, Ord, Read, Show)
```

```
*Main> :k Either
Either :: * -> * -> *
```

Конструктор з двома параметрами. Треба з одним

У конструктора (Either a) - тільки один параметр

```
*Main> :k Either String
Either String :: * -> *
```

```
Int
Right Int
> (+) (Right 1) (Right 2)
```

- []
- Maybe
- IO
- Either a

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

- []
- Maybe
- IO
- Either a

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

insert :: (Ord a) => a -> Tree a -> Tree a
insert x EmptyTree = Node x EmptyTree EmptyTree
insert x (Node y left right)
                  | x == y = Node x left right
                  | x <  y = Node y (insert x left) right
                  | x >  y = Node y  left (insert x right)
```

- []
- Maybe
- IO
- Either a

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

insert :: (Ord a) => a -> Tree a -> Tree a
insert x EmptyTree = Node x EmptyTree EmptyTree
insert x (Node y left right)
                  | x == y = Node x left right
                  | x <  y = Node y (insert x left) right
                  | x >  y = Node y  left (insert x right)

*Tree> foldr (insert) EmptyTree [5,1,4,9,8,2,3]
Node 3 (Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree)
(Node 8 (Node 4 EmptyTree (Node 5 EmptyTree EmptyTree)) (Node 9 EmptyTree EmptyTree))
```

- []
- Maybe
- IO
- Either a

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


*Tree> :k Tree
Tree :: * -> *
```

- []
- Maybe
- IO
- Either a

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


*Tree> :k Tree
Tree :: * -> *


Int
Tree Int
(+) :: Int -> Int -> Int

> 1 + 2
> (+) (Node 1 EmptyTree EmptyTree) (Node 2 EmptyTree EmptyTree)
```

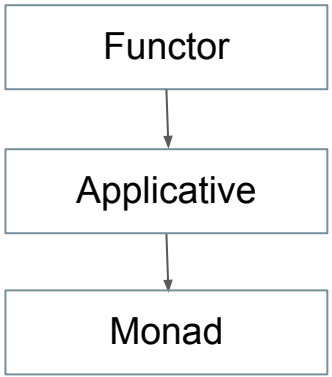
- []
- Maybe
- IO
- Either a
- Tree

Кажуть про:

контейнерні типи;

обчислювальний контекст

- [] недетермінований результат
- Maybe з можливим неуспіхом
- IO з можливим побічним ефектом
- Either a з можливим неуспіхом та повідомленням
- Tree ієрархічна структура



Клас Functor

class Functor f where

Клас Functor використовується для типів, які передбачають відображення (mapped over).

Екземпляри (втілення) функтора (Functor) повинні задовольняти наступним законам:

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

Functors in "Categories for Programmers", Bartosz Milewski

Клас Functor

Minimal complete definition: fmap

fmap :: (a -> b) -> f a -> f b
метод 1 арг. 2 арг. результат

1 арг.: функція a -> b
2 арг.: тип *a* в функторній обгортці/контексті *f*, а саме *f a*
результат: тип *b* в функторній обгортці *f*, а саме *f* b

Метод отримує функцію типу a -> b та значення типу a в функторній обгортці, повертає значення типу b в тій же функторній обгортці.

Функторна обгортка — конструктор типу (з одним параметром)