

Курс: Функциональное программирование
Практика 7. Аппликативные функторы и свёртки

Аппликативные функторы

► Устно вычислите значения выражений и проверьте результат в GHCi:

```
[(*0),(+100),(^2)] <*> [1,2,3]
(++> <$> ["ha","heh","hmm"]) <*> ["?","!","..."]
[(+),(*)] <*> [1,2] <*> [3,4]
getZipList $ (,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
(,) <$> "dog" <*> "cat" <*> "rat"
```

Напишем представителя аппликативного функтора для ((->) a)

```
instance Applicative ((->) e) where
  pure      = const
  (<*>) f g x = ????????????
```

Попробуем записать тип

```
(<*>) :: f (a -> b) -> f a -> f b
      == (e -> (a -> b)) -> (e -> a) -> (e -> b)
      == (e -> a -> b) -> (e -> a) -> e -> b
```

- Что это за функция?
- Каков тип следующих конструкций для ((->) a)

```
\f g h -> f <*> g <*> h
\f g h -> f <$> g <*> h
```

► Устно вычислите значения выражений и проверьте результат в GHCi:

```
(+) <*> (*3) $ 4
(+) <$> (+2) <*> (*3) $ 10
(\a b c -> [a,b,c]) <$> (+5) <*> (*3) <*> (/2) $ 7
```

► Напишите

```
instance Applicative (Either e) where
  pure          =
  (<*>)         =
```

Моноиды

Моноид — это множество с ассоциативной бинарной операцией над ним (`mappend`) и единицей для этой операции (`mempty`).

Эндоморфизм (стрелка из типа в него же)

```
newtype Endo a = Endo { appEndo :: a -> a }
```

образует моноид относительно композиции.

► Напишите

```
instance Monoid (Endo a) where
  mempty = ???
  Endo f 'mappend' Endo g = ???
```

Свёртки

► Устно вычислите значение выражения и проверьте результат в GHCi:

```
foldr (/) 2 [8,12,24,4]
```

► Напишите реализацию следующих функций через свёртки `foldr` или `foldr1`

```
head' :: [a] -> a
head'  =
```

```
tail' :: [a] -> [a]
tail'  =
```

```
or' :: [Bool] -> Bool
or'  =
```

```
maximum' :: Ord a => [a] -> a
maximum'  =
```

```
map' :: (a -> b) -> [a] -> [b]
map' f  =
```

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p  =
```

► Напишите две реализации `reverse :: [a] -> [a]` — через свёртки `foldr` и `foldl`

```
reverse' =
```

```
reverse'' =
```

► Напишите реализацию функции `foldTree` — свёртки для двоичного дерева

```
data Tree a = Nil | Branch (Tree a) a (Tree a)
```

► Используя `foldTree`, напишите функцию `flattenTree :: Tree a -> [a]`.
Сколько разных версий такой функции Вы можете написать?

► Попробуйте написать

```
instance Applicative Tree where
  pure          =
  (<*>)         =
```

Сравните результат с `instance Applicative []`.

► Имеется семейство функций `zipWith`, `zipWith3`, `zipWith4...`

```
*Test> let x1s = [1,2,3]
*Test> let x2s = [4,5,6]
*Test> let x3s = [7,8,9]
*Test> let x4s = [10,11,12]
*Test> zipWith (\a b -> 2*a+3*b) x1s x2s
[14,19,24]
*Test> zipWith3 (\a b c -> 2*a+3*b+5*c) x1s x2s x3s
[49,59,69]
*Test> zipWith4 (\a b c d -> 2*a+3*b+5*c-4*d) x1s x2s x3s x4s
[9,15,21]
```

Апликативные функторы могут заменить всё это семейство

```
*Test> getZipList $ (\a b -> 2*a+3*b) <$> ZipList x1s <*> ZipList x2s
[14,19,24]
*Test> getZipList $ (\a b c -> 2*a+3*b+5*c) <$> ZipList x1s <*> ZipList x2s <*> ZipList x3s
[49,59,69]
*Test> getZipList $ (\a b c d -> 2*a+3*b+5*c-4*d) <$> ZipList x1s <*> ZipList x2s <*>
ZipList x3s <*> ZipList x4s
[9,15,21]
```

Реализуйте операторы `(>*<)` и `(>$<)`, позволяющие спрятать упаковку `ZipList` и распаковку `getZipList`:

```
*Test> (\a b -> 2*a+3*b) >$< x1s >*< x2s  
[14,19,24]  
*Test> (\a b c -> 2*a+3*b+5*c) >$< x1s >*< x2s >*< x3s  
[49,59,69]  
*Test> (\a b c d -> 2*a+3*b+5*c-4*d) >$< x1s >*< x2s >*< x3s >*< x4s  
[9,15,21]
```