Курс: Функциональное программирование
Практика 8. Аппликативные функторы

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)`, и как их можно было бы записать, не используя аппликативный стиль:

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

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
(pure 3) "blah"

(+) <*> (*3) $ 4

(+) <$> (+2) <*> (*3) $ 10

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

▶ Напишите

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

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
(*) <$> Right 2 <*> Right 3

(*) <$> Right 2 <*> Left "Oh."

(*) <$> Left "Ha!" <*> Left "Oh."

(*) <$> Left "Ha!" <*> undefined
```

Аппликативный интерфейс библиотеки Parsec

```
> import Control.Applicative ((<*>), (*>), (<$>), (<*))
> import Text.Parsec

> let prs rule text =  parse rule "" text
> prs letter "ABC"
Right 'A'
> prs digit "ABC"
Left (line 1, column 1):
unexpected "A" expecting digit

> prs (many1 digit) "123"
Right "123"
> prs (many1 digit) "123ABC"
Right "123"

> let p0 = (,) <$> many1 letter <*> many1 digit
> prs p0 "ABC123"
Right ("ABC","123")

> prs (string "ABC" *> many digit) "ABC132"
Right "132"
> prs (string "ABC" *> many digit) "ABD132"
Left (line 1, column 1):
unexpected "D"
expecting "ABC"

> let p1 = (,) <$> many1 letter <*> (many space *> many1 digit)
> prs p1 "ABC    123"
Right ("ABC","123")
```

```
> let p2 = string "ABC" <|> string "DEF"
> prs p2 "ABC123"
Right "ABC"
> prs p2 "DEF123"
Right "DEF"
> prs p2 "GHI123"
Left (line 1, column 1):
unexpected "G" expecting "ABC" or "DEF"
```

► Что вернут следующие вызовы?

```
prs (char 'A' <* char 'B' <* char 'C') "ABC"

prs (char 'A' *> char 'B' <* char 'C') "ABC"

prs (char 'A' *> char 'B' *> char 'C') "ABC"

prs (char 'A' <* char 'B' *> char 'C') "ABC"
```

Класс `Traversable`

► Устно вычислите значения выражений и проверьте результат в GHCi:

```
sequenceA [Right 3,Right 4,Right 5]

sequenceA [Right 3,Left 4,Right 5]

sequenceA [Left 3,Left 4,Right 5]

sequenceA [Right 3,Left 4,undefined]

sequenceA [undefined,Left 4,Right 5]

traverse (\x -> if odd x then Right x else Left x) [1,3,5,7]

traverse (\x -> if odd x then Right x else Left x) [1,2,6,7]

sequenceA [(+3),(+2),(+1)] 3

sequenceA [("cc",2),("dd",3),("ee",5)]

sequenceA [[1,2,3],[4,5,6]]

(getZipList . sequenceA . map ZipList) [[1,2,3],[4,5,6]]
```

► Введём следующий трёхпараметрический типовый оператор, инкапсулирующий композицию однопараметрических конструкторов типов:

```
newtype Cmps f g x = Cmps {getCmps :: f (g x)}
```

Каков кайнд этого конструктора типов? Приведите пример замкнутого типа, сконструированного с помощью `Cmps`, и пример терма этого типа.
▶ Определите функцию

```
ffmap h = getCmps . fmap h . Cmps
```

и объясните её выведенный тип. Попробуйте осуществить вызов

```
> ffmap (+42) $ Just [1,2,3]
```

В чём причина ошибки?
▶ Чтобы обеспечить работоспособность подобного вызова, сделайте тип `Cmps` представителем класса типов `Functor`:

```
instance (Functor f, Functor g) => Functor (Cmps f g) where
  fmap                =   ???
```

Проверьте работоспособность на примерах:

```
> ffmap (+42) $ Just [1,2,3]
Just [43,44,45]

> ffmap (+42) $ [Just 1,Just 2,Nothing]
[Just 43,Just 44,Nothing]
```

Нетрудно проверить, что все законы функторов выполняются для этого представителя. Таким образом **композиция функторов является функтором**. Подобное утверждение верно для `Applicative`, `Foldable`, `Traversable`, но неверно для `Monad`.

▶ Проверьте, что законы аппликативных функторов выполняются для `Maybe` и списков.

Домашнее задание

▶ (1 балл) В модуле `Data.List` имеется семейство функций `zipWith`, `zipWith3`, `zipWith4`, ...:

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

Аппликативные функторы могут заменить всё это семейство

```
GHCi> getZipList $ (\a b -> 2*a+3*b) <$> ZipList x1s <*> ZipList x2s
[14,19,24]
GHCi> getZipList $ (\a b c -> 2*a+3*b+5*c) <$> ZipList x1s <*> ZipList x2s <*> ZipList x3s
[49,59,69]
GHCi> 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`:

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

► (1 балл) Следующий тип данных задает гомогенную тройку элементов, которую можно рассматривать как трехмерный вектор:

```
data Triple a = Tr a a a deriving (Eq,Show)
```

Сделайте этот тип функтором и аппликативным функтором с естественной для векторов семантикой, подобной `ZipList`.

```
GHCi> (^2) <$> Tr 1 (-2) 3
Tr 1 4 9
GHCi> Tr (^2) (+2) (*3) <*> Tr 2 3 4
Tr 4 5 12
```

► (1 балл) Сделайте тип `Triple` из предыдущего задания представителем классов типов `Foldable` и `Traversable`:

```
GHCi> Data.Foldable.foldl (++) "!!" (Tr "ab" "cd" "efg")
"!!abcdefg"
GHCi> traverse (\x -> if x>10 then Right x else Left x) (Tr 12 14 16)
Right (Tr 12 14 16)
GHCi> traverse (\x -> if x>10 then Right x else Left x) (Tr 12 8 4)
Left 8
```

► (1 балл) Сделайте двоичное дерево

```
data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)
```

функтором и аппликативным функтором, реализовав в последнем случае семантику применения узла к соответствующему узлу второго дерева:

```
GHCi> let t1 = Branch (Branch Nil 1 Nil) 2 Nil
GHCi> let t2 = Branch (Branch Nil 3 Nil) 4 (Branch Nil 5 Nil)
GHCi> (*) <$> t1 <*> t2
Branch (Branch Nil 3 Nil) 8 Nil
```

▶ (2 балла) Сделайте тип `Tree` из предыдущего задания представителем класса типов `Traversable`:

```
GHCi> traverse (\x -> if odd x then Right x else Left x) (Branch (Branch Nil 1
 Nil) 3 Nil)
Right (Branch (Branch Nil 1 Nil) 3 Nil)
GHCi> traverse (\x -> if odd x then Right x else Left x) (Branch (Branch Nil 1
 Nil) 2 Nil)
Left 2
```

▶ (2 балла) Сделайте тип

```
newtype Cmps f g x = Cmps {getCmps :: f (g x)}
```

представителем класса типов `Applicative`.

▶ (1 балл) Сделайте тип

```
newtype Cmps f g x = Cmps {getCmps :: f (g x)}
```

представителем класса типов `Foldable`.

▶ (1 балл) Сделайте тип

```
newtype Cmps f g x = Cmps {getCmps :: f (g x)}
```

представителем класса типов `Traversable`.