Курс: Функциональное программирование
Практика 15. Молнии (зипперы), линзы и прочая оптика

Зипперы

На лекции мы, продифференцировав тип дерева

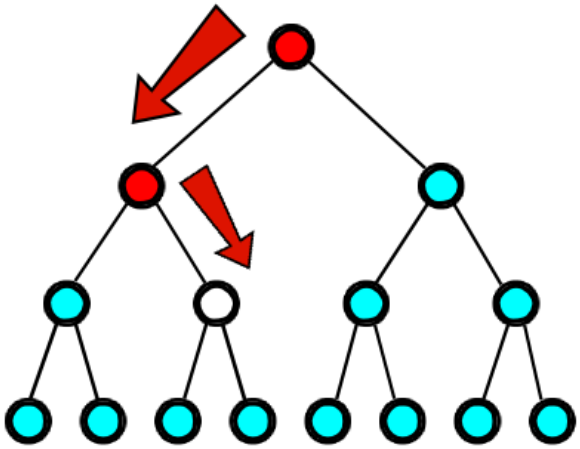
```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

выяснили как должен быть устроен его контекст с дыркой:

$$\mathsf{T}(\alpha) = 1 + \alpha * \mathsf{T}^2(\alpha)$$

$$\begin{aligned}
&\mathsf{T}'(\alpha) = \mathsf{T}^2(\alpha) + \alpha * 2 * \mathsf{T}(\alpha) * \mathsf{T}'(\alpha) \\
&\mathsf{T}'(\alpha)(1 - 2 * \alpha * \mathsf{T}(\alpha)) = \mathsf{T}^2(\alpha) \\
&\mathsf{T}'(\alpha) = \mathsf{T}^2(\alpha)/(1 - 2 * \alpha * \mathsf{T}(\alpha)) \\
&\mathsf{T}'(\alpha) = \mathsf{T}^2(\alpha) * \mathsf{L}(2 * \alpha * \mathsf{T}(\alpha))
\end{aligned}$$

$$\mathsf{T}'(\alpha) = \mathsf{T}(\alpha) * \mathsf{T}(\alpha) * \mathsf{L}(2 * \alpha * \mathsf{T}(\alpha))$$



Производная $\mathsf{T}'(\alpha) = \mathsf{T}(\alpha) * \mathsf{T}(\alpha) * \mathsf{L}(2 * \alpha * \mathsf{T}(\alpha))$ задает факторизацию:
$\mathsf{T}(\alpha) * \mathsf{T}(\alpha)$ – два поддерева ниже фокуса;
$\mathsf{L}(2 * \alpha * \mathsf{T}(\alpha))$ = `[(Bool,a,Tree a)]`, где
`Bool` — указывает идти налево или направо;
`a` — значение родительского узла;
`Tree a` — второе поддерево родительского узла.

► (2 балла) Реализуйте зиппер для дерева, используя изложенные выше соображения.

```
type TreeZ a = (a, CntxT a)

data CntxT a = CntxT (Tree a) (Tree a) [(Dir, a, Tree a)]
  deriving (Eq, Show)

data Dir = L | R deriving (Eq, Show)
```

(Здесь введен тип `Dir`, изоморфный `Bool`, но имеющий содержательные с точки зрения использования имена конструкторов.)

Напишите функции для создания зиппера (`mktz`), навигации по нему (`left`, `right`, `up`), обратному превращению в дерево (`untz`) и изменения значения в фокусе (`updTZ`):

```
mktz :: Tree a -> TreeZ a
mktz = undefined

left :: TreeZ a -> TreeZ a
left = undefined

right :: TreeZ a -> TreeZ a
right = undefined

up :: TreeZ a -> TreeZ a
up = undefined

untz :: TreeZ a -> Tree a
untz  = undefined

updTZ :: a -> TreeZ a -> TreeZ a
updTZ  = undefined
```

Предполагается, что деревья непустые, и пользователь, передвигаясь по зипперу, сам следит за тем, чтобы фокус не выходил за пределы дерева. Иными словами никакой обработки ошибок реализовывать не требуется.

Тесты используют дерево

```
  2
 / \
1   4
   / \
  3   5

GHCi> tr = Node (Node Empty 1 Empty) 2 (Node (Node Empty 3 Empty) 4 (Node Empty 5 Empty))
GHCi> mktz tr
```

```
(2,CntxT (Node Empty 1 Empty) (Node (Node Empty 3 Empty) 4 (Node Empty 5 Empty)) [])

GHCi> (left . mktz) tr
(1,CntxT Empty Empty [(L,2,Node (Node Empty 3 Empty) 4 (Node Empty 5 Empty))])
GHCi> (left . right . mktz) tr
(3,CntxT Empty Empty [(L,4,Node Empty 5 Empty),(R,2,Node Empty 1 Empty)])

GHCi> (up . left . mktz) tr == mktz tr
True
GHCi> (up $ right $ mktz $ tr) == mktz tr
True
GHCi> (up . up . left . right . mktz) tr == mktz tr
True

GHCi> import Data.Function ((&))
GHCi> tr & mktz & right & left & updTZ 42 & untz
Node (Node Empty 1 Empty) 2 (Node (Node Empty 42 Empty) 4 (Node Empty 5 Empty))
```

Линзы ван Лаарховена

Линза — инструмент для манипулирования подструктурой типа `a` некоторой структуры данных типа `s`, находящейся в «фокусе» этой линзы.
На лекции мы строили линзы ван Лаарховена

```
type Lens s a = forall f. Functor f => (a -> f a) -> s -> f s
```

Упаковка геттера и сеттера в такую линзу:

```
lens :: (s -> a) -> (s -> a -> s) -> Lens s a
lens get set = \ret s -> fmap (set s) (ret $ get s)
```

Например, линзы для пары строится так:

```
_1 :: Lens (a,b) a
_1 = lens (\(x,_) -> x) (\(_,y) v -> (v,y))

_2 :: Lens (a,b) b
_2 = lens (\(_,y) -> y) (\(x,_) v -> (x,v))
```

Распаковка осуществляется с помощью конкретных функторов. Геттер вынимается с помощью функтора `Const`:

```
newtype Const a x = Const {getConst :: a}

instance Functor (Const c) where
  fmap _ (Const v) = Const v

view :: Lens s a -> s -> a
view lns s = getConst (lns Const s)
```

Сеттер вынимается с помощью функтора `Identity`:

```
newtype Identity a = Identity {runIdentity :: a}

instance Functor Identity where
  fmap f (Identity x) = Identity (f x)

over :: Lens s a -> (a -> a) -> s -> s
over lns fn s = runIdentity $ lns (Identity . fn) s

set :: Lens s a -> a -> s -> s
set lns a s = over lns (const a) s
```

Полиморфные линзы для кортежей

Линзы `_1` и `_2` разумно использовать не только для пар, но и для кортежей больших размеров:

```
GHCi> view _1 ('a',12,False)
12
GHCi> over _2 succ (True,41,"ABC")
(True,42,"ABC")
```

Этого можно добиться, реализовав эти линзы не как свободные функции, а как элементы классов типов:

```
class Field1 s a where
  _1 :: Lens s a

class Field2 s a where
  _2 :: Lens s a

class Field3 s a where
  _3 :: Lens s a
```

и так далее.
► (1 балл) Реализуйте представителей этих классов типов для доступа к элементам двух- и трехэлементных кортежей. (Подумайте, не нужны ли в этих классах типов функциональные зависимости.)

```
GHCi> over _1 (^2) (3,True,"ABC")
(9,True,"ABC")
GHCi> over _3 tail (3,True,"ABC")
(3,True,"BC")
```

Обобщение базовых линз ван Лаарховена

Библиотека `lens` работает с более общим понятием линзы:

```
type Lens s t a b = forall f. Functor f => (a -> f b) -> s -> f t
```

При этом простые линзы, с которыми мы имели дело раньше, в `lens` называются `Lens'`

```
type Lens' s a = Lens s s a a
```

Обобщенные линзы позволяют изменять тип структуры при ее модификации:

```
GHCi> set _2 "ABC" (5,7)
(5,"ABC")
GHCi> over _1 length ("ABC",True)
(3,True)
```

▶ (1 балл) Измените определения функций `lens`, `_1` и `_2` (используйте свободные версии для пар), `view`, `set` и `over` так, чтобы подобная модификация типа структуры была возможной.

```
GHCi> over (_2 . _1) length (4,("ABCDE",True))
(4,(5,True))
GHCi> over (_1 . _2) odd (("ABC",42),13)
(("ABC",False),13)
```

▶ (* балл) Попробуйте теперь реализовать `_1`, `_2` и `_3`, как элементы четырехпараметрических классов типов, так чтобы они могли обслуживать не только пары, но и тройки.

```
GHCi> over (_2 . _1) length (4,("ABCDE",True))
(4,(5,True))
GHCi> over (_2 . _1) length (4,("ABCDE",True), 33)
(4,(5,True),33)
GHCi> over (_3 . _2) length (1,2,(False,"ABCDE"))
(1,2,(False,5))
```

Библиотека `lens`

▶ Используя Template Haskell изготовьте оптику (вызывая `makeLenses`) для типа данных

```
data EitherOrOr a b c
  = Lft { _lft :: a }
  | Mid { _mid :: b }
  | Rgh { _rgh :: c }
  deriving (Eq,Show)
```

Попробуйте устно определить, что вернут следующие вызовы, и затем проверьте себя в GHCi:

```
GHCi> :i rgh

GHCi> v = (Mid 'A',42)
GHCi> v & _1 . mid .~ 'B'

GHCi> v & _1 . mid .~ True

GHCi> v & _1 . lft .~ 'B'

GHCi> v ^? _1 . mid

GHCi> v ^?! _1 . mid

GHCi> v ^? _1 . rgh

GHCi> v ^?! _1 . rgh
```

▶ Добавьте теперь дополнительно вызов `makePrisms` для типа `EitherOrOr`. Попробуйте устно определить, что вернут следующие вызовы, и затем проверьте себя в GHCi:

```
GHCi> :i _Rgh

GHCi> v = (Mid 'A',42)
GHCi> v & _1 . _Mid .~ 'B'

GHCi> v & _1 . _Mid .~ True

GHCi> v & _1 . _Lft .~ 'B'

GHCi> v ^? _1 . _Mid
```

```
GHCi> v ^?! _1 . _Mid

GHCi> v ^? _1 . _Rgh

GHCi> v ^?! _1 . _Rgh
```