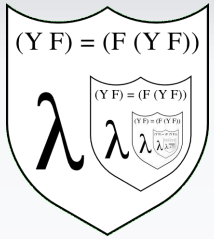
Парсеры. Естественные преобразования

Функциональное программирование

2023 г.



Контейнер класса парсер

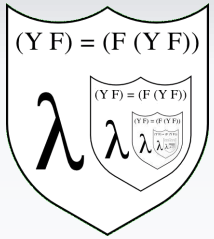
```
newtype Parser a = P (String -> [(a,String)])
parse :: Parser a -> String -> [(a,String)]
parse (P p) inp = p inp
```

Простейший парсер

```
item :: Parser Char
item = P (\inp -> case inp of
    [] -> []
    (x:xs) -> [(x,xs)])
```

Парсер как функтор

```
instance Functor Parser where
- fmap :: (a -> b) -> Parser a -> Parser b
 fmap g p = P (\inp -> case parse p inp of
     [] -> []
     [(v,out)] -> [(g v,out)])
```



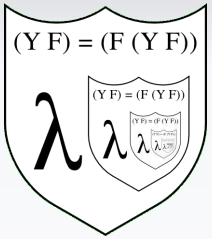
Два определения fmap

1

```
instance Functor Parser where
- fmap :: (a -> b) -> Parser a -> Parser b
 fmap f p = Parser (\inp -> case parse p inp of
     [] -> []
     [(v,out)] -> [(f v,out)])
```

2

```
instance Functor Parser where
  fmap :: (a -> b) -> Parser a -> Parser b
 fmap f (Parser p1) = Parser
     (\ s ->
         map (\(s, val) -> (s, f val)) (p1 s))
```



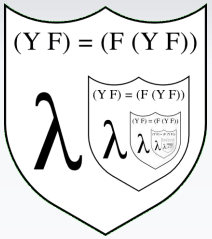
Парсеры

Два примера

```
prefixP :: String -> Parser String
prefixP s = Parser f
  where
    f input = if pref s input
        then [(drop (length s) input, s)]
        else []
skipString :: String -> Parser ()
skipString s = Parser f
 where
    f input = if pref s input
        then [(drop (length s) input, ())]
        else []
```

Альтернатива:

```
skipString s = () <$ prefixP s
```

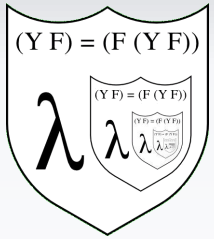


Аппликативные парсеры

Чтобы распарсить несколько элементов подряд, выгодно использовать аппликативное представление.

Парсер как аппликатив

```
instance Applicative Parser where
-- pure :: a -> Parser a
 pure v = P (\inp -> [(v,inp)])
-- <*> :: Parser (a -> b) -> Parser a -> Parser b
 pg <*> px = P (\inp -> case parse pg inp of
     [] -> []
     [(g,out)] -> parse (fmap g px) out)
```



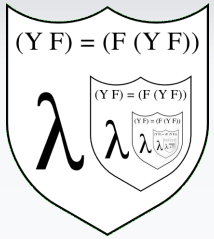
Два определения аппликации

1

```
instance Applicative Parser where
 pure v = P (\inp -> [(v,inp)])
 pg <*> px = P (\inp -> case parse pg inp of
     [] -> []
     [(g,out)] -> parse (fmap g px) out)
```

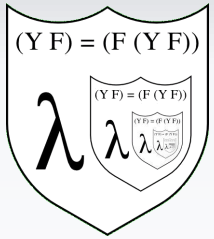
2

```
instance Applicative Parser where
 pure x = Parser (\s -> [(s, x)])
 pf <*> px = Parser (\s ->
     [ (sx, f x) | (sf, f) <- parse pf $ s,
                   (sx, x) <- parse px $ sf ] )
```



Комбинации парсеров

```
data MyStructType = MyStruct
 { field1 :: Type1
 , field2 :: Type2
 , field3 :: Type3}
parserAll :: Parser MyStructType
parserAll = MyStruct
        <$> parser1 <*> parser2 <*> parser3
```



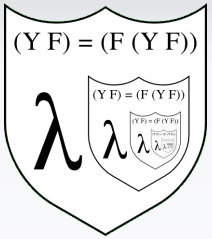
Парсер альтернативы

Определение

```
class Applicative f => Alternative f where
 empty :: f a
 (<|>) :: f a -> f a -> f a
instance Alternative Parser where
  empty = Parser (const [])
  px <|> py =
       Parser (\s -> parse px s ++ parse py s)
```

Стандартные функции

```
some v = (:) <$> v <*> many v
many v = some v <|> pure []
```

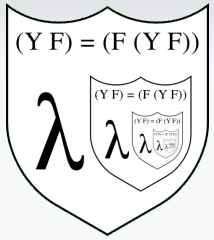


Пример применения

Зависит от того, допускает ли парсер неоднозначный разбор.

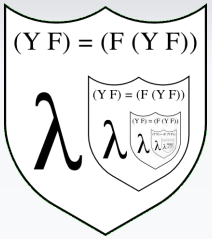
Две записи

```
-- Первый вариант
fmap f (Parser p)
    = Parser (\b -> map (second f) (p b))
-- Второй вариант
fmap f (Parser p1) = Parser
    (\ s ->
        map (\(s, val) -> (s, f val)) (p1 s))
```

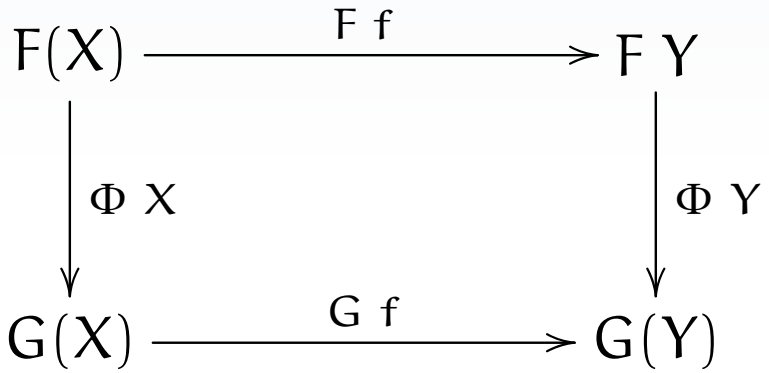


Применение парсеров

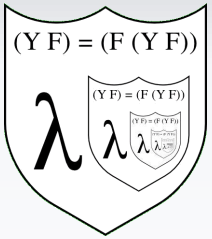
```
parseString :: String -> Parser a -> Maybe a
parseString s (Parser p) = case p s of
    [("", val)] -> Just val
    _           -> Nothing
predP :: (Char -> Bool) -> Parser Char
predP p = Parser f
 where
    f "" = []
    f (c : cs) | p c = [(cs, c)]
               | otherwise = []
stringP :: String -> Parser String
stringP s = Parser f
 where
    f s' | s == s' = [("", s)]
         | otherwise = []
```



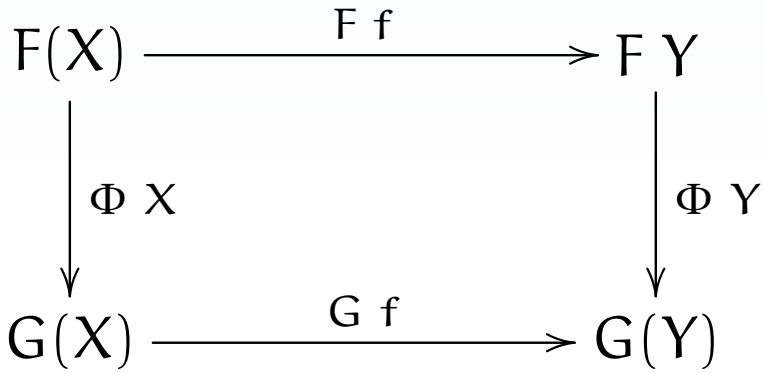
Естественное преобразование



Естественные преобразования — полиморфные функции.



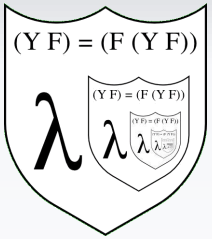
Естественное преобразование



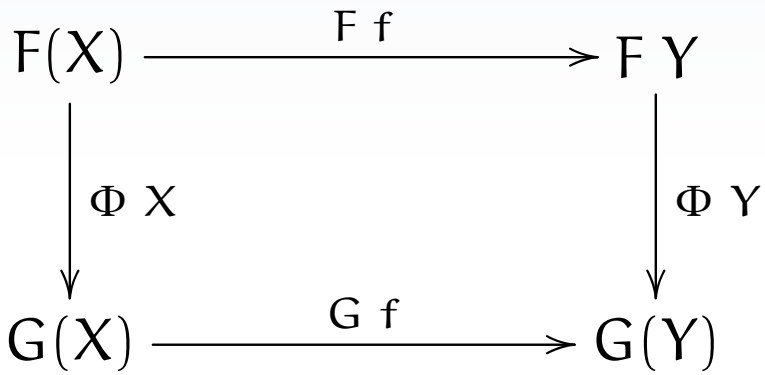
Естественные преобразования — полиморфные функции.

Закон ЕП

```
fmap f.nu = nu. fmap f
```



Естественное преобразование



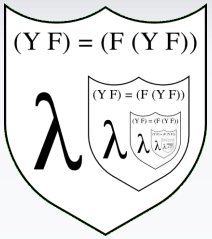
Естественные преобразования — полиморфные функции.

Закон ЕП

```
fmap f.nu = nu. fmap f
```

Пример

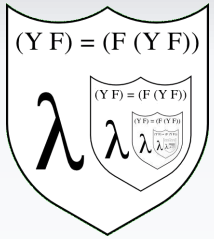
```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```



Еще пример естественного преобразования

Функтор Const

```
newtype Const b a = Const { getConst :: b }
instance Functor (Const b) where
  fmap _ (Const x) = Const x
```

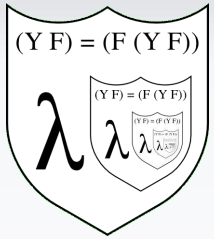


Еще пример естественного преобразования

Функтор Const

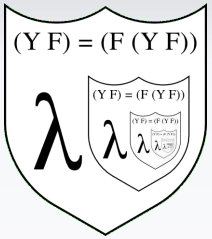
```
newtype Const b a = Const { getConst :: b }
instance Functor (Const b) where
  fmap _ (Const x) = Const x

-- length как естественное преобразование
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + getConst (length xs))
```



Задача 1

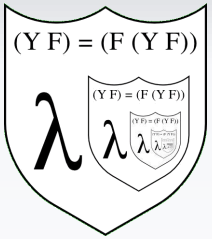
```
deleteDN (Not (Not x)) = deleteDN x
deleteDN (Not x) = Not $ deleteDN x
deleteDN (Or y z) = Or (deleteDN y) (deleteDN z)
deleteDN x = x
```



Задача 2

```
deleteTaut (Or x y) | triv w v = ValT
                    | otherwise = w ‘Or‘ v
                            where w = deleteTaut x
                                  v = deleteTaut y
deleteTaut (Not x) | v == ValT = ValF
                   | v == ValF = ValT
                   | otherwise = Not v
                     where v = deleteTaut x
deleteTaut x = x

triv ValT x = True
triv x ValT = True
triv x (Not y) | x == y = True
               | otherwise = False
triv (Not x) y | x == y = True
               | otherwise = False
triv _ _ = False
```

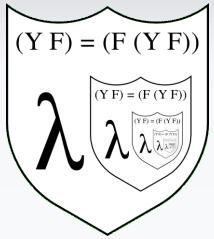


Еще раз о составных типах

- type — задает синоним типа.
- newtype — задает новый тип в упаковке однопараметрического конструктора.
- data — аналог newtype, но конструкторы могут быть разными.

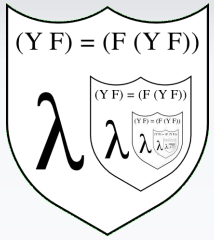
После компиляции newtype упаковка уничтожается. После компиляции data — нет, поэтому используются методы извлечения из-под конструктора (runIdentity, parse, ...).

```
newtype LazyType = Lazy Int
f (Lazy n) = True
> f undefined
True
```



Определение собственных методов

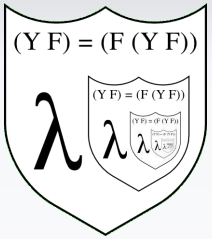
```
instance (Eq a) => Eq [a] where
    [] == [] = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _ == _ = False
```



Определение собственных методов

```
instance (Eq a) => Eq [a] where
    [] == [] = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _ == _ = False
```

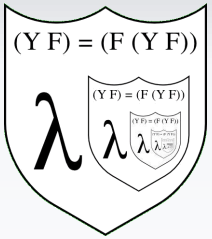
```
data MyEither = R a | L a
instance (Show a) => Show (MyEither a) where
  show (R a) = "Error in the expression "++ show a
  show (L a) = show a
```



Пример

Предположим, мы хотим написать вывод лямбда-выражений в привычной форме.

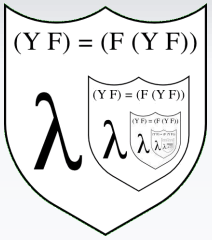
```
data Le a = VarExpr a | Lambb a (Le a) | App (Le a) (Le a)
instance (Show a) => Show (Le a) where
    show (VarExpr x) = ...
    show (Lambb x y) = ...
    show (App x y) = ...
```



Полиморфизм и отношения

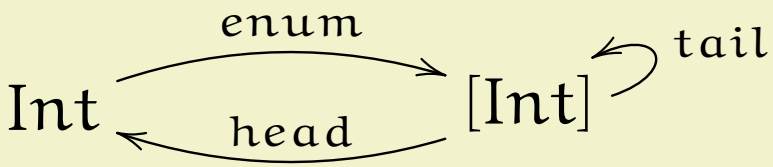
(Reynolds) Типы — множества отношений. Мономорфные типы — множества вида $\{(x_i, x_i)\}$ (отношения только тривиальные).

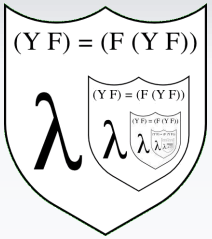
Функции — частный случай отношений (в обычном смысле теории множеств). Параметрически полиморфная функция сохраняет отношения между типами.



Задачи

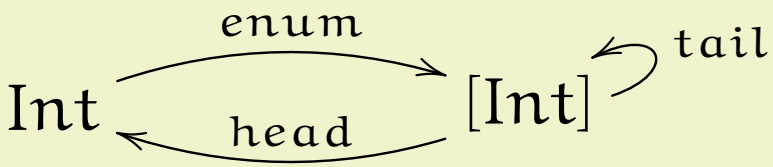
Какие морфизмы есть в этих категориях? И есть ли естественные преобразования?

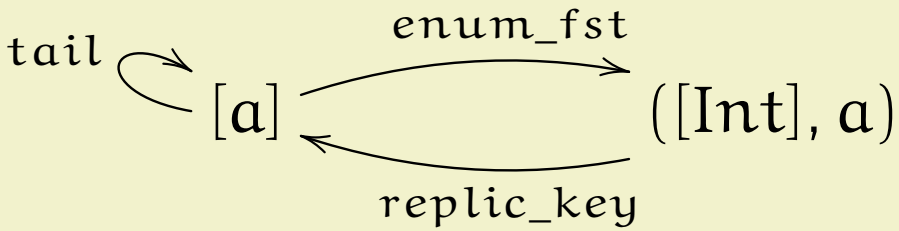


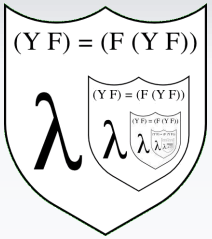


Задачи

Какие морфизмы есть в этих категориях? И есть ли естественные преобразования?

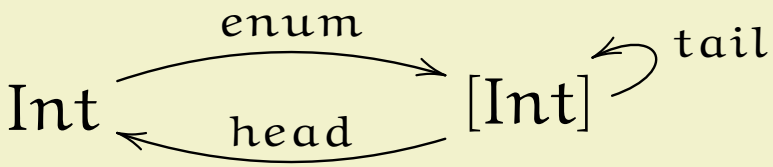


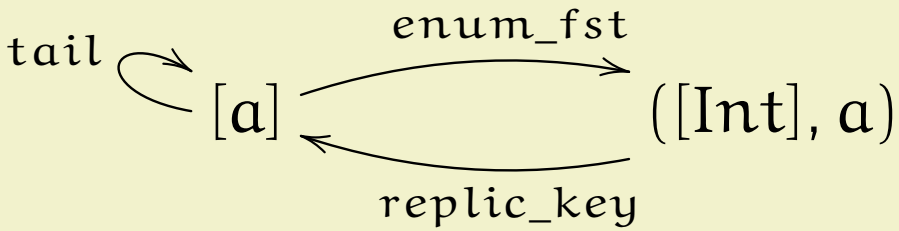


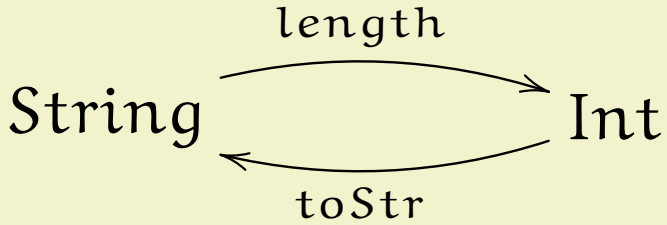


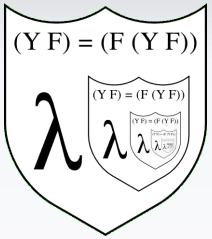
Задачи

Какие морфизмы есть в этих категориях? И есть ли естественные преобразования?







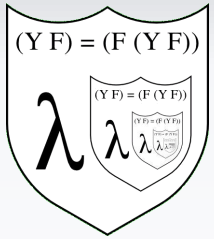


Theorems for free

Предложение

Параметрически полиморфные функции — естественные преобразования.

Благодаря этому свойству, можно строить утверждения о всех функциях, имеющих данный полиморфный тип.



Theorems for free

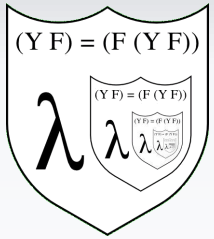
Предложение

Параметрически полиморфные функции — естественные преобразования.

Благодаря этому свойству, можно строить утверждения о всех функциях, имеющих данный полиморфный тип.

Примеры

- reverse.map = map.reverse



Theorems for free

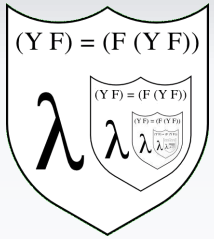
Предложение

Параметрически полиморфные функции — естественные преобразования.

Благодаря этому свойству, можно строить утверждения о всех функциях, имеющих данный полиморфный тип.

Примеры

- reverse.map = map.reverse
- length.map = length



Theorems for free

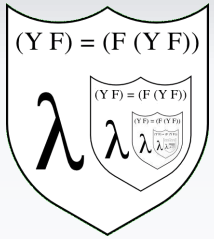
Предложение

Параметрически полиморфные функции — естественные преобразования.

Благодаря этому свойству, можно строить утверждения о всех функциях, имеющих данный полиморфный тип.

Примеры

- reverse.map = map.reverse
- length.map = length
- f.head = head.fmap f



Theorems for free

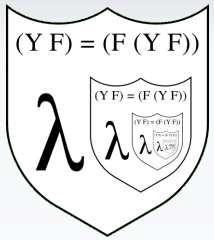
Предложение

Параметрически полиморфные функции — естественные преобразования.

Благодаря этому свойству, можно строить утверждения о всех функциях, имеющих данный полиморфный тип.

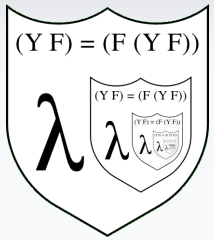
Примеры

- reverse.map = map.reverse
- length.map = length
- f.head = head.fmap f
- fmap f.tail = tail.fmap f



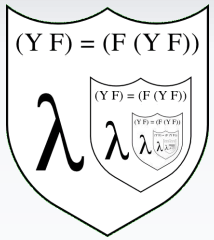
Однозначность определений полиморфных функций

1. Как зависит от второго аргумента полиморфная функция типа String $\to$ a $\to$ String?



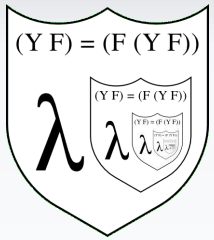
Однозначность определений полиморфных функций

1. Как зависит от второго аргумента полиморфная функция типа String $\to$ a $\to$ String?
2. Что должна возвращать на Nothing функция типа Maybe a $\to$ [a]?



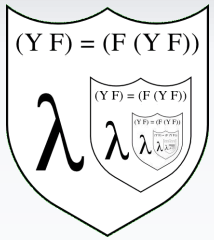
Однозначность определений полиморфных функций

1. Как зависит от второго аргумента полиморфная функция типа String $\to$ a $\to$ String?
2. Что должна возвращать на Nothing функция типа Maybe a $\to$ [a]?
3. Какие существуют функции типа Maybe a $\to$ Maybe a?



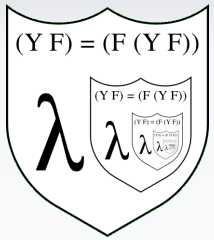
Однозначность определений полиморфных функций

1. Как зависит от второго аргумента полиморфная функция типа String $\to$ a $\to$ String?
2. Что должна возвращать на Nothing функция типа Maybe a $\to$ [a]?
3. Какие существуют функции типа Maybe a $\to$ Maybe a?
4. Пусть функция f имеет тип [a] $\to$ String. Верно ли, что f (x ++ y) = f(x) + f(y)?



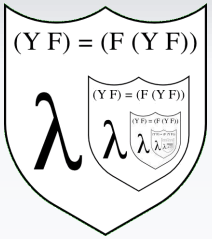
Однозначность определений полиморфных функций

1. Как зависит от второго аргумента полиморфная функция типа String $\to$ a $\to$ String?
2. Что должна возвращать на Nothing функция типа Maybe a $\to$ [a]?
3. Какие существуют функции типа Maybe a $\to$ Maybe a?
4. Пусть функция f имеет тип [a] $\to$ String. Верно ли, что f (x ++ y) = f(x) + f(y)?
5. Верно ли, что replicate [a] — это функция типа [a] $\to$ Int $\to$ [[a]]?



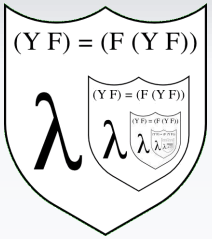
Однозначность определений полиморфных функций

1. Как зависит от второго аргумента полиморфная функция типа String $\to$ a $\to$ String?
2. Что должна возвращать на Nothing функция типа Maybe a $\to$ [a]?
3. Какие существуют функции типа Maybe a $\to$ Maybe a?
4. Пусть функция f имеет тип [a] $\to$ String. Верно ли, что f (x ++ y) = f(x) + f(y)?
5. Верно ли, что replicate [a] — это функция типа [a] $\to$ Int $\to$ [[a]]? Привести пример такой функции.



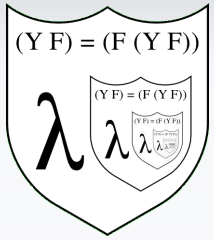
Композиционность

- Функторы `f . g` — это просто `fmap . fmap`. По определению, всегда тоже функтор;
- Аппликативы — конструкция `((<*> <$> f <*> x))`. Тоже всегда аппликатив.
- Монады??? Не всегда можно скомбинировать. Пример — `Maybe IO`.



Трансформеры монад

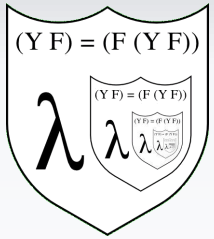
- Нужно определить `return` для композиции (тривиально);
- Нужно определить `bind`: заходим во внешнюю монаду с помощью её `bind`-оператора и осуществляем обработку внутренней монады;
- Нужно определить способ «поднять» значение из внешней монады в композицию: оператор `lift`.



Пример

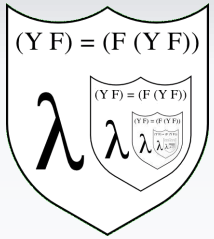
```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
MaybeT :: m (Maybe a) -> MaybeT m a
runMaybeT :: MaybeT m a -> m (Maybe a)
return :: a -> MaybeT m a
return = lift . return
(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
mx >>= k = MaybeT $ do -- заходим в монаду m
v <- runMaybeT mx
case v of
Nothing -> return Nothing
Just y -> runMaybeT (k y)

-- Класс типов <<трансформер монад>>
instance MonadTrans MaybeT where
lift :: m a -> MaybeT m a
lift = MaybeT . liftM Just
```



Законы трансформеров

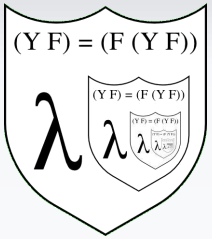
- `lift . return == return`
- `lift (m >>= k) == lift m >>= (lift . k)`



Зипперы

- Как сделать списки, потенциально бесконечные с двух сторон? Возможное решение: хранить начало (инвертированное) и конец списка.
- Возникает структура `zipper`, состоящая из «зерна», начала и хвоста.

```
type ListZipper a = (a, ([a], [a]))
make :: [a] -> ListZipper a
make (x : xs) = (x, ([], xs))
fwd (e, (xs, (y:ys))) = (y, ((e:xs), ys))
bwd (e, ((x:xs), ys)) = (x, (xs, (e:ys)))
unzip (x, ([],xs)) = (x : xs)
unzip (x, lists) = unzip.bwd (x, lists)
```

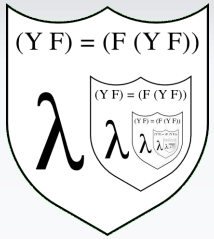


Комонады

- Из зиппера всегда можно извлечь значение, и зиппер можно передать в следующий уровень вложенности. Типы соответствующих операций: `z a -> a`, `z a -> z (z a)`. Это почти монадические операции `return` и `join`, только наоборот. Получается структура комонады.
- Реализация для списков:

```
extract :: w a -> a
extract (a, lists) = a

duplicate :: w a -> w (w a)
duplicate (a, (before, after))
    = ((a, (before, after)),
       (deepen_back before (a:after),
        deepen_forward (a:before) after)))
```



Законы комонад

```
extract . duplicate      = id
fmap extract . duplicate = id
duplicate . duplicate    = fmap duplicate . duplicate
```