Функциональное программирование Лекция 9. Использование аппликативных функторов

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

06.11.2025

План лекции

1. Аппликативные парсеры
2. Класс типов `Alternative`
3. Класс типов `Traversable`

План лекции

Парсеры

- ***Парсер*** — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противоположном случае.
- Простейший, но неудобный и не полностью соответствующий определению парсер:
  `type Parser a = String -> a`

Парсеры

- ***Парсер*** — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противоположном случае.
- Простейший, но неудобный и не полностью соответствующий определению парсер:
  `type Parser a = String -> a`
- Версия 2 (храним неразобранный остаток):
  `type Parser a = String -> (String,a)`

Парсеры

- ***Парсер*** — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противоположном случае.
- Простейший, но неудобный и не полностью соответствующий определению парсер:
  ```
  type Parser a = String -> a
  ```
- Версия 2 (храним неразобранный остаток):
  ```
  type Parser a = String -> (String,a)
  ```
- Версии 3,4 (умеем обрабатывать ошибки):
  ```
  type Parser a = String -> Maybe (String,a)
  type Parser a = String -> Either String (String,a)
  ```

Парсеры

- ***Парсер*** — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противоположном случае.
- Простейший, но неудобный и не полностью соответствующий определению парсер:
  `type Parser a = String -> a`
- Версия 2 (храним неразобранный остаток):
  `type Parser a = String -> (String,a)`
- Версии 3,4 (умеем обрабатывать ошибки):
  `type Parser a = String -> Maybe (String,a)`
  `type Parser a = String -> Either String (String,a)`
- Версия 5 (неоднозначные грамматики):
  `type Parser a = String -> [(String,a)]`

Парсеры: конструируем тип

- Выберем в качестве базовой Версию 3:
  ```
  type Parser a = String -> Maybe (String,a)
  ```
- Возможное обобщение — абстрагироваться по типу входного потока:
  ```
  type Parser tok a = [tok] -> Maybe ([tok],a)
  ```

Парсеры: конструируем тип

- Выберем в качестве базовой Версию 3:
  `type Parser a = String -> Maybe (String,a)`
- Возможное обобщение — абстрагироваться по типу входного потока:
  `type Parser tok a = [tok] -> Maybe ([tok],a)`
- Итоговая версия, годная для реализации представителей:

```
newtype Parser tok a =
  Parser { runParser :: [tok] -> Maybe ([tok],a) }
```

Парсер в работе

```haskell
newtype Parser tok a =
  Parser { runParser :: [tok] -> Maybe ([tok],a) }
```

```haskell
charA :: Parser Char Char
charA = Parser f where
  f (c:cs) | c == 'A' = Just (cs,c)
  f _                 = Nothing
```

```
GHCi> runParser charA "ABC"
Just ("BC",'A')
GHCi> runParser charA "BCD"
Nothing
```

Функции для конструирования парсеров

```
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy pr = Parser f where
  f (c:cs) | pr c  = Just (cs,c)
  f _              = Nothing
```

```
GHCi> runParser (satisfy isUpper) "ABC"
Just ("BC",'A')
GHCi> runParser (satisfy isLower) "ABC"
Nothing
```

```
lower :: Parser Char Char
lower = satisfy isLower

char :: Char -> Parser Char Char
char c = satisfy (== c)
```

Парсер как функтор

```haskell
digit :: Parser Char Int
digit = satisfy isDigit     -- returns Char :(
```

Парсер как функтор

```haskell
digit :: Parser Char Int
digit = satisfy isDigit     -- returns Char :(
```

```haskell
digit :: Parser Char Int
digit = digitToInt <$> satisfy isDigit
```

Парсер как функтор

```haskell
digit :: Parser Char Int
digit = satisfy isDigit     -- returns Char :(
```

```haskell
digit :: Parser Char Int
digit = digitToInt <$> satisfy isDigit
```

Для этого нужно сделать парсер функтором:

```haskell
instance Functor (Parser tok) where
  fmap :: (a -> b) -> Parser tok a -> Parser tok b
  fmap g (Parser p) = Parser f where
    f xs = case p xs of
      Just (cs, c) -> Just (cs, g c)
      Nothing      -> Nothing
```

Парсер как функтор (2)

Предыдущее объявление функтора может быть записано компактнее.

```
newtype Parser tok a =
  Parser { runParser :: [tok] ->  Maybe ([tok],a) }
```

Действительно, тип `Parser` — не что иное, как последовательная композиция трех функторов: `(->) [tok]`, `Maybe` и `(,) [tok]`. Поэтому

```
instance Functor (Parser tok) where
  fmap :: (a -> b) -> Parser tok a -> Parser tok b
  fmap g = Parser . (fmap . fmap . fmap) g . runParser
```

Парсер как функтор (2)

Предыдущее объявление функтора может быть записано компактнее.

```
newtype Parser tok a =
  Parser { runParser :: [tok] ->  Maybe ([tok],a) }
```

Действительно, тип `Parser` — не что иное, как последовательная композиция трех функторов: `(->) [tok]`, `Maybe` и `(,) [tok]`. Поэтому

```
instance Functor (Parser tok) where
  fmap :: (a -> b) -> Parser tok a -> Parser tok b
  fmap g = Parser . (fmap . fmap . fmap) g . runParser
```

или (указав громоздкие типы, см. код к лекции)

```
instance Functor (Parser tok) where
  fmap = coerce (fmap . fmap . fmap :: ...)
```

Аппликативный парсер: pure

```
instance Applicative (Parser tok) where
  pure :: a -> Parser tok a
  pure x = Parser $ \s -> Just (s, x)
  ...
```

СЕМАНТИКА: Результат `pure` — переданное в него значение; входная строка не потребляется.

```
GHCi> runParser (pure 42) "Answer"
Just ("Answer",42)
```

«Непотребление» входной строки обеспечит безэффектность реализации `pure`.

Аппликативный парсер: (<*>)

```
(<*>) :: Parser tok (a -> b) ->
         Parser tok a -> Parser tok b
Parser u <*> Parser v = Parser f where
  f xs = case u xs of
    Nothing       -> Nothing
    Just (xs', g) -> case v xs' of
      Nothing         -> Nothing
      Just (xs'', x) -> Just (xs'', g x)
```

СЕМАНТИКА: получить результат первого парсера, затем второго на остатке строки, и применить первый ко второму. Неудача хотя бы одного парсера приводит к тотальной неудаче.

```
GHCi> runParser (pure (,) <*> digit <*> digit) "12AB"
Just ("AB",(1,2))
GHCi> runParser ((,) <$> digit <*> digit) "1AB2"
Nothing
```

Аппликативный парсер, попытка другой реализации

Можно попытаться определить аппликативный функтор как композицию трех аппликативных функторов

```
instance Applicative (Parser tok) where
  pure  = Parser . pure . pure . pure
  Parser u <*> Parser v =
      Parser $ (liftA2 . liftA2) (<*>) u v
```

Это настоящий композитный аппликативный функтор, но он не обладает нужной нам семантикой

```
GHCi> runParser (pure 42) "Answer"
Just ("",42)
GHCi> runParser (pure (,) <*> digit <*> digit) "12AB"
Just ("2AB2AB",(1,1))
```

Пример использования аппликативного интерфейса

Теперь можем строить «сложные» парсеры:

```
multiplication :: Parser Char Int
multiplication = (*) <$> digit <* char '*' <*> digit
```

Все операторы левоассациотивны и имеют один (4) приоритет.

```
GHCi> runParser multiplication "6*7"
Just ("",42)
```

А как сделать универсальный парсер, годный для разбора строк вида "63*796"?

Пример использования аппликативного интерфейса

Теперь можем строить «сложные» парсеры:

```
multiplication :: Parser Char Int
multiplication = (*) <$> digit <* char '*' <*> digit
```

Все операторы левоассациотивны и имеют один (4) приоритет.

```
GHCi> runParser multiplication "6*7"
Just ("",42)
```

А как сделать универсальный парсер, годный для разбора строк вида `"63*796"`?
Написать рекурсивный парсер.

```
digits0 :: Parser Char [Int]
digits0 = (:) <$> digit <*> digits0
```

К сожалению, одним (`<*>`) не обойтись: неудача `digit` на `'*'` уничтожит весь разбор.

План лекции

1. Аппликативные парсеры
2. Класс типов `Alternative`
3. Класс типов `Traversable`

Класс Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
infixl 3 <|>
```

Операция `<|>` задает моноид над `f a` независимым от `a` образом, наделяя «умножающий» аппликативный функтор дополнительным «сложением».

```
instance Alternative [] where
  empty :: [a]
  empty  = []
  (<|>) :: [a] -> [a] -> [a]
  (<|>)   = (++)
```

Представитель `Alternative` для списков полностью повторяет определение моноида для списка.

Alternative vs Monoid

```haskell
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> y       = y
  x       <> Nothing = x
  Just x  <> Just y  = Just (x <> y)

instance Semigroup a => Monoid (Maybe a) where
  mempty = Nothing
```

```haskell
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
instance ??? => Alternative Maybe where ???
```

С `Alternative` невозможно наложить ограничение на параметр `Maybe` на уровне класса типов для реализации полиморфного представителя.

Представитель Alternative для Maybe

```
instance Alternative Maybe where
  empty :: Maybe a
  empty  =  Nothing

  (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> m  =  m
  m       <|> _  =  m
```

Представитель `Alternative` для `Maybe` ведёт себя, как упаковка `First`, возвращая первый не-`Nothing` в цепочке альтернатив:

```
GHCi> Nothing <|> Just 3 <|> Just 5 <|> Nothing
Just 3
```

Представитель Alternative для ZipList

```
instance Alternative ZipList where
  empty :: Maybe a
  empty = ZipList []

  (<|>) :: ZipList a -> ZipList a -> ZipList a
  ZipList xs <|> ZipList ys =
      ZipList (xs ++ drop (length xs) ys)
```

Если первый кончился, а второй еще нет — дополняем первый остатком второго.

```
GHCi> ZipList "abc" <|> ZipList "ABCDEFG"
ZipList {getZipList = "abcDEFG"}
```

Интерфейс Alternative для парсера

```
instance Alternative (Parser tok) where
  empty :: Parser tok a
  empty = Parser $ \_ -> Nothing

  (<|>) :: Parser tok a -> Parser tok a -> Parser tok a
  Parser u <|> Parser v = Parser f where
    f xs = case u xs of
      Nothing -> v xs
      z       -> z
```

СЕМАНТИКА:

- `empty` — парсер, всегда возвращающий неудачу;
- `(<|>)` — пробуем первый, при неудаче пробуем второй на исходной строке.

Пример использования альтернативного интерфейса

```
GHCi> runParser  (char 'A' <|> char 'B') "ABC"
Just ("BC",'A')
GHCi> runParser  (char 'A' <|> char 'B') "BCD"
Just ("CD",'B')
```

Теперь можем сделать «хороший» рекурсивный парсер

```
lowers :: Parser Char String
lowers = (:) <$> lower <*> lowers <|> pure ""
```

```
GHCi> runParser lowers "abCd"
Just ("Cd","ab")
GHCi> runParser lowers "abcd"
Just ("","abcd")
GHCi> runParser lowers "Abcd"
Just ("Abcd","")
```

Полное определение класса Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  some, many :: f a -> f [a]
  some v = (:) <$> v <*> many v      -- One or more
  many v = some v <|> pure []        -- Zero or more

optional :: Alternative f => f a -> f (Maybe a)
optional v = Just <$> v <|> pure Nothing
```

```
lowers :: Parser Char String
lowers = many lower
digits :: Parser Char [Int]
digits = many digit
```

Примеры для some, many и optional

```
GHCi> runParser (many digit) "42abdef"
Just ("abdef",[4,2])
GHCi> runParser (some digit) "42abdef"
Just ("abdef",[4,2])

GHCi> runParser (many digit) "abdef"
Just ("abdef",[])
GHCi> runParser (some digit) "abdef"
Nothing

GHCi> runParser (optional digit) "42abdef"
Just ("2abdef",Just 4)
GHCi> runParser (optional digit) "abdef"
Just ("abdef",Nothing)
```

План лекции

1. Аппликативные парсеры
2. Класс типов `Alternative`
3. Класс типов `Traversable`

Аппликативный дистрибьютор списка

```
dist :: Applicative f => [f a] -> f [a]
dist []       = pure []
dist (ax:axs) = pure (:) <*> ax <*> dist axs
```

Функция `pure` поднимает конструкторы; `ax :: f a`, `dist axs :: f [a]`, поэтому все можно сцеплять `<*>`.

```
GHCi> dist [Just 3,Just 5]
Just [3,5]
GHCi> dist [Just 3,Nothing]
Nothing
GHCi> getZipList $ dist $ map ZipList [[1,2],[3,4],[5,6]]
[[1,3,5],[2,4,6]]
```

Использование в теле определения `dist` тех же самых конструкторов, что и в образцах, приводит к сохранению трехэлементной структуры списка.

Аппликативный дистрибьютор списка (2)

```
dist :: Applicative f => [f a] -> f [a]
dist []       = pure []
dist (ax:axs) = (:) <$> ax <*> dist axs
```

```
GHCi> getZipList $ dist $ map ZipList [[1,2],[3,4],[5,6]]
[[1,3,5],[2,4,6]]
```

Что будет, если убрать `ZipList`, оставив просто список?

Аппликативный дистрибьютор списка (2)

```
dist :: Applicative f => [f a] -> f [a]
dist []       = pure []
dist (ax:axs) = (:) <$> ax <*> dist axs
```

```
GHCi> getZipList $ dist $ map ZipList [[1,2],[3,4],[5,6]]
[[1,3,5],[2,4,6]]
```

Что будет, если убрать ZipList, оставив просто список?

```
GHCi> dist [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2
,4,6]]
```

И здесь трехэлементная структура внешнего списка сохранилась во внутреннем.

Класс типов Traversable

Минимальное полное определение: `traverse` или `sequenceA`.

```
class (Functor t, Foldable t) => Traversable t where
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse g = sequenceA . fmap g
```

`sequenceA`: обеспечиваем правило коммутации нашего функтора `t` с произвольным аппликативным функтором `f`. Структура внешнего контейнера `t` сохраняется, а апликативные эффекты внутренних `f` объединяются в результирующем `f`. `traverse` — это `fmap` с эффектами: проезжаем по структуре `t a`, последовательно применяя функцию к элементам типа `a` и монтируем в точности ту же структуру из результатов типа `b`, параллельно «коллекционируя» эффекты.

Представители класса типов Traversable

```
instance Traversable Maybe where
  traverse :: Applicative f =>
                  (a -> f b) -> Maybe a -> f (Maybe b)
  traverse _ Nothing  = pure Nothing
  traverse g (Just x) = Just <$> g x
```

```
instance Traversable [] where
  traverse :: Applicative f =>
                    (a -> f b) -> [a] -> f [b]
  traverse _ []     = pure []
  traverse g (x:xs) = (:) <$> g x <*> traverse g xs
```

```
GHCi> traverse (\x -> [x+10,x+100]) (Just 7)
[Just 17,Just 107]
GHCi> traverse (\x -> [x+10,x+100]) [7,8]
[[17,18],[17,108],[107,18],[107,108]]
```

Сравнение реализаций Traversable и Functor

```
instance Traversable Maybe where
  traverse _ Nothing  = pure Nothing
  traverse g (Just x) = pure Just <*> g x

instance Functor     Maybe where
  fmap     _ Nothing  =      Nothing
  fmap     g (Just x) =      Just    (g x)
```

```
instance Traversable [] where
  traverse _ []     = pure []
  traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs

instance Functor     [] where
  fmap _ []         =      []
  fmap g (x:xs)     =      (:)    (g x)   (fmap     g xs)
```

Первый закон Traversable

Имеется стандартный контейнер, слишком простой, чтобы иметь эффекты.

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

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

instance Applicative Identity where
  pure = Identity
  Identity g <*> Identity x = Identity (g x)
```

(1) identity

```
traverse Identity  ≡  Identity
```

```
GHCi> traverse Identity [1,2,3]
Identity [1,2,3]
```

Реализация fmap по умолчанию

Всякий Traversable — это Functor: имея traverse мы можем универсальным образом реализовать fmap, удовлетворяющий законам функтора.

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault g = runIdentity . traverse (Identity . g)
```

```
Identity  ::  b -> Identity b

Identity . g  ::  a -> Identity b

traverse (Identity . g)  ::  t a -> Identity (t b)

runIdentity . traverse (Identity . g)  ::  t a -> t b
```

Имеется также foldMapDefault для реализации Foldable.

Законы Traversable (2) и (3)

(2) composition

```
traverse (Compose . fmap f . g)  ≡
   Compose . fmap (traverse f) . traverse g
```

Здесь обе части имеют тип `t a -> Compose g f (t c)` в предположении, что `g :: a -> g b` и `f :: b -> f c`.

(3) naturality

```
h . traverse f  ≡  traverse (h . f)
```

где `h :: (Applicative f, Applicative g) => f b -> g b` произвольный *аппликативный гомоморфизм*, то есть функция удовлетворяющая требованиям:
(1) `h (pure x) = pure x`;
(2) `h (x <*> y) = h x <*> h y`.
В предположении, что `f :: a -> f b`, обе части имеют тип `t a -> g (t b)`.

Законы Traversable: практический смысл

Законы `Traversable` дают следующие гарантии:

- Траверсы не пропускают элементов.
- Траверсы посещают элементы не более одного раза.
- `traverse pure` дает `pure`.
- Траверсы не изменяют исходную структуру — она либо сохраняется, либо полностью исчезает.

```
GHCi> traverse Just [1,2,3]
Just [1,2,3]
GHCi> traverse (const Nothing) [1,2,3]
Nothing
GHCi> traverse (const Nothing) [1,undefined]
Nothing
```