ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ*, ПРОЕКТИРОВАНИЕ И РЕФАКТОРИНГ*, ПРОГРАММИРОВАНИЕ*, JAVASCRIPT*, БЛОГ КОМПАНИИ MAIL.RU GROUP

Функциональное программирование в JavaScript с практическими примерами

ПЕРЕВОД

AloneCoder 28 апреля 2017 в 14:20 34,2k

Оригинал: Raja Rao DV

Функциональное программирование (ФП) может улучшить ваш подход к написанию кода. Но ФП непросто освоить. Многие статьи и руководства не уделяют внимания таким подробностям, как монады (Monads), аппликативность (Applicative) и т. д., не приводят в качестве иллюстраций практические примеры, которые могли бы

помочь нам в повседневном использовании мощных ФП-методик. Я решил исправить это упущение.

Хочу подчеркнуть: в статье сделан упор на том, **ЗАЧЕМ** нужна фича X, а не на том, **ЧТО** такое фича X.

Функциональное программирование

ФП — это стиль написания программ, при котором просто комбинируется набор функций. В частности, ФП подразумевает обёртывание в функции практически всего подряд. Приходится писать много маленьких многократно используемых функций и вызывать их одну за другой, чтобы получить результат вроде **(func1.func2.func3)** или комбинации типа **func1(func2(func3()))**.

Но чтобы действительно писать программы в таком стиле, функции должны следовать определённым правилам и решать некоторые проблемы.

Проблемы ФП

Если всё можно сделать путём комбинирования набора функций, то…

1. Как обрабатывать условие if-else? **(Подсказка: монада Either)**
2. Как обрабатывать исключения Null? **(Подсказка: монада Maybe)**
3. Как удостовериться, что функции действительно многократно используемые и могут использоваться везде? **(Подсказка:**

чистые функции (Pure functions), ссылочная прозрачность (referential transparency))

4. Как удостовериться, что данные, передаваемые нами в функции, не изменены и могут использоваться и в других местах? **(Подсказка: чистые функции (Pure functions), неизменяемость)**
5. Если функция берёт несколько значений, но при объединении в цепочку (chaining) можно передавать только по одной функции, то как нам сделать эту функцию частью цепочки? **(Подсказка: каррирование (currying) и функции высшего порядка)**
6. И многое другое <добавьте сюда ваш вопрос>.

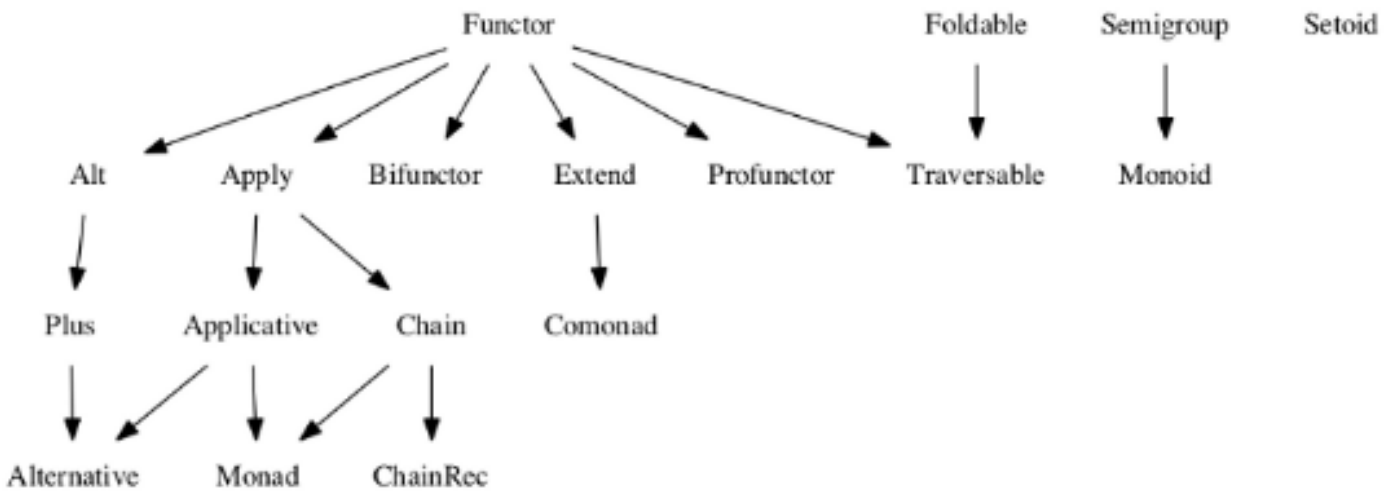
ФП-решение

Для решения всех этих проблем полностью функциональные языки вроде Haskell из коробки предоставляют разные инструменты и математические концепции, например монады, функторы и т. д. JavaScript из коробки не даёт такого обилия инструментов, но, к счастью, у него достаточный набор ФП-свойств, позволяющих писать библиотеки.

Спецификации Fantasy Land и ФП-библиотеки

Если библиотеки хотят предоставить такие возможности, как функторы, монады и пр., то им нужно реализовать функции/классы, удовлетворяющие определённым спецификациям, чтобы предоставляемые возможности были такими же, как в языках вроде Haskell.

Яркий пример — спецификации Fantasy Land, объясняющие, как должна себя вести каждая JS-функция/класс.



На иллюстрации изображены все спецификации и их зависимости. Спецификации — по сути законы, они аналогичны интерфейсам в Java. С точки зрения JS спецификации можно рассматривать как классы или функции-конструкторы, реализующие в соответствии со спецификацией некоторые методы (вроде **map, of, chain** и т. д.).

Например:

JS-класс — это функтор (Functor), если он реализует метод map. И метод должен работать так, как предписано спецификацией (объяснение упрощённое, правил на самом деле больше).

JS-класс — это функтор Apply (Apply Functor), если он в соответствии со спецификацией реализует функции map и ap.

JS-класс — это монада (Monad Functor), если он реализует требования Functor, Apply, Applicative, Chain и самой Monad (в

соответствии с цепочкой зависимостей).

Примечание: зависимость может выглядеть как наследование, но необязательно. Например, монада реализует обе спецификации — Applicative и Chain (в дополнение к остальным).

Библиотеки, совместимые со спецификациями Fantasy Land

Есть несколько библиотек, реализующих спецификации FL. Например: **monet.js**, **barely-functional**, **folktalejs**, **ramda-fantasy (на базе Ramda)**, **immutable-ext (на базе ImmutableJS)**, **Fluture и др.**

Какие библиотеки мне лучше использовать?

Библиотеки наподобие lodash-fp и ramdajs позволят только начать писать в стиле ФП. Но они не предоставляют функции для использования ключевых математических концепций вроде монад, функторов или редьюсера (Foldable), позволяющих решать реальные проблемы.

Так что я бы вдобавок порекомендовал выбрать одну из библиотек, использующих спецификации FL: **monet.js**, **barely-functional**, **folktalejs**, **ramda-fantasy (на базе Ramda)**, **immutable-ext (на базе ImmutableJS)**, **Fluture и т. д.**

Примечание: я пользуюсь ramdajs и ramda-fantasy.

Итак, мы получили представление об основах, теперь перейдём к практическим примерам и изучим различные возможности и методики ФП.

Пример 1. Работа с проверками на Null

В разделе рассматриваются: функторы, монады, монады Maybe, каррирование

Применение: мы хотим показывать разные начальные страницы в зависимости от пользовательской настройки **предпочтительного языка**. Нужно написать **getUrlForUser**, возвращающий соответствующий URL из списка URL'ов **(indexURL'ы)** для пользовательского **(joeUser)** предпочтительного языка (испанский).

Проблема: язык не может быть null. И пользователь тоже не может быть null (не залогинен). Языка может не быть в нашем списке indexURL'ов. Поэтому нам нужно позаботиться о многочисленных nulls или undefined.

```
//TODO Напишите это в императивном и функциональном стилях
const getUrlForUser = (user) => {
//todo
}
//Пользовательский объект
let joeUser = {
    name: 'joe',
    email: 'joe@example.com',
    prefs: {
        languages: {
            primary: 'sp',
            secondary: 'en'
        }
    }
};
//Глобальная схема indexURL'ов для разных языков
let indexURLs = {
    'en': 'http://mysite.com/en',  //Английский
     'sp': 'http://mysite.com/sp', //Испанский
    'jp': 'http://mysite.com/jp'   //Японский
}
//apply url to window.location
const showIndexPage = (url) => { window.location = url };
```

Решение (императивное против функционального):

Не переживайте, если ФП-версия выглядит трудной для понимания. Дальше в этой статье мы разберём её шаг за шагом.

```
//Императивная версия:
//Слишком много if-else и проверок на null; зависимость от
глобальных indexURL'ов; «английские» URL'ы берутся для всех стран по
умолчанию

const getUrlForUser = (user) => {
  if (user == null) { //не залоггирован
    return indexURLs['en']; //возвращает страницу по умолчанию
  }
  if (user.prefs.languages.primary && user.prefs.languages.primary
!= 'undefined') {
    if (indexURLs[user.prefs.languages.primary]) {//если есть
```

```
локализованная версия, то возвращает
indexURLs[user.prefs.languages.primary];
    } else {
      return indexURLs['en'];
    }
  }
}

//вызов
showIndexPage(getUrlForUser(joeUser));

//Функциональное программирование:
//(Сначала сложно для понимания, но получается надёжнее, багов
меньше)
//Используемые ФП-методики: функторы, монада Maybe и Currying
const R = require('ramda');
const prop = R.prop;
const path = R.path;
const curry = R.curry;
const Maybe = require('ramda-fantasy').Maybe;

const getURLForUser = (user) => {
    return Maybe(user)//обёртываем пользователя в объект Maybe
        .map(path(['prefs', 'languages', 'primary'])) //используем
Ramda для получения первичного языка
        .chain(maybeGetUrl); //передаём язык в maybeGetUrl и
получаем URL или монаду null
}

const maybeGetUrl = R.curry(function(allUrls, language) {//
преобразуем в функцию с одним аргументом
    return Maybe(allUrls[language]);//возвращаем монаду (url | null)
})(indexURLs);// вместо глобального доступа передаём indexURLs

function boot(user, defaultURL) {
   showIndexPage(getURLForUser(user).getOrElse(defaultURL));
}

boot(joeUser, 'http://site.com/en'); //'http://site.com/sp'
```

Давайте сначала разберём ФП-концепции и методики, использованные в этом решении.

Функторы

Любой класс (или функция-конструктор) или тип данных, хранящий значение и реализующий метод map, называется функтором (Functor).

Например: массив — это функтор, потому что он может хранить значения и имеет метод map, позволяющий нам применить (map) функцию к хранимым значениям.

```
const add1 = (a) => a+1;
let myArray = new Array(1, 2, 3, 4); //хранимые значения
myArray.map(add1) // -> [2,3,4,5] //применение функций
```

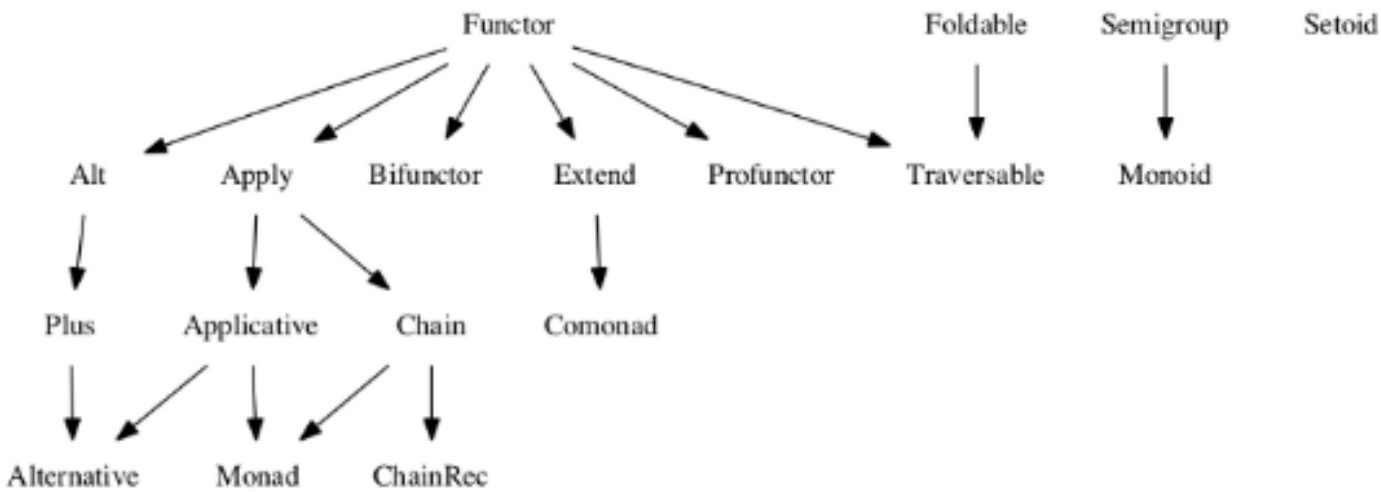
Напишем собственный функтор — MyFunctor. Это просто JS-класс (функция-конструктор), который хранит какое-то значение и реализует метод map. Этот метод применяет функцию к хранимому значению, а затем из результата создаёт новый Myfunctor и возвращает его.

```
const add1 = (a) => a + 1;
class MyFunctor { //кастомный функтор
  constructor(value) {
    this.val = value;
  }
  map(fn) {   //применяет функцию к this.val + возвращает новый
Myfunctor
   return new Myfunctor(fn(this.val));
  }
}
//temp – это экземпляр функтора, хранящий значение 1
let temp = new MyFunctor(1);
temp.map(add1) //-> temp позволяет нам преобразовать (map) "add1"
```

P. S. Функторы должны реализовывать и другие спецификации (Fantasy-land) в дополнение к map, но здесь мы этого касаться не будем.

Монады

Монады тоже функторы, т. е. у них есть метод **map**. Но они реализуют не только его. Если вы снова взглянете на схему зависимостей, то увидите, что монады должны реализовывать разные функции из разных спецификаций, например Apply (метод **ap**), Applicative (методы **ap** и **of**) и Chain (метод **chain**).



***Упрощённое объяснение**. В JS монады — это классы или функции-конструкторы, хранящие какие-то данные и реализующие методы map, ap, of и chain, которые что-то делают с хранимыми данными в соответствии со спецификациями.*

Вот образец реализации, чтобы вы понимали внутреннее устройство монад.

```
//Монада - образец реализации
class Monad {
    constructor(val) {
        this.__value = val;
    }
    static of(val) {//Monad.of проще, чем new Monad(val)
        return new Monad(val);
    };
    map(f) {//Применяет функцию, но возвращает другую монаду!
        return Monad.of(f(this.__value));
    };
    join() { // используется для получения значения из монады
        return this.__value;
    };
    chain(f) {//вспомогательная функция, преобразующая (map), а
затем извлекающая значение
        return this.map(f).join();
    };
     ap(someOtherMonad) {//используется для работы с несколькими
монадами
        return someOtherMonad.map(this.__value);
    }
}
```

Обычно используются монады не общего назначения, а более специфические и более полезные. Например, Maybe или Either.

Монада Maybe

Монада Maybe — это класс, реализующий спецификацию монады. Но особенность монады в том, что она корректно обрабатывает значения null или undefined.

В частности, если хранимые данные являются null или undefined, то функция map вообще не выполняет данную функцию, потому не возникает проблем с null и undefined.

Такая монада используется в ситуациях, когда приходится иметь дело с null-значениями.

В коде ниже представлена ramda-fantasy реализация монады Maybe. В зависимости от значения она создаёт экземпляр одного из двух разных подклассов — **Just** или **Nothing** (значение или полезное, или null/undefined).

Хотя методы **Just** и **Nothing** одинаковы (map, orElse и т. д.), Just что-то делает, а Nothing не делает ничего.

Обратите особое внимание на методы map и orElse в этом коде:

```
//Здесь приведены фрагменты реализации Maybe из библиотеки ramda-fantasy
//Полный код доступен по ссылке: https://github.com/ramda/ramda-fantasy/blob/master/src/Maybe.js

function Maybe(x) { //<-- главный конструктор, возвращающий монаду Maybe подкласса Just или Nothing
  return x == null ? _nothing : Maybe.Just(x);
}

function Just(x) {
  this.value = x;
}
util.extend(Just, Maybe);

Just.prototype.isJust = true;
Just.prototype.isNothing = false;

function Nothing() {}
util.extend(Nothing, Maybe);

Nothing.prototype.isNothing = true;
Nothing.prototype.isJust = false;

var _nothing = new Nothing();
```

```
Maybe.Nothing = function() {
  return _nothing;
};

Maybe.Just = function(x) {
  return new Just(x);
};

Maybe.of = Maybe.Just;

Maybe.prototype.of = Maybe.Just;

// функтор
Just.prototype.map = function(f) { //Делает map, когда Just
выполняет функцию, и возвращает Just на основании результата
  return this.of(f(this.value));
};

Nothing.prototype.map = util.returnThis; // <-- Делает map, когда
Nothing ничего не делает

Just.prototype.getOrElse = function() {
  return this.value;
};

Nothing.prototype.getOrElse = function(a) {
  return a;
};

module.exports = Maybe;
```

Давайте посмотрим, как можно использовать монаду Maybe для работы с проверками на null.

Пойдём поэтапно:

1. Если есть какой-то объект, который может быть null или иметь null-свойства, то создаём из него объект-монаду.
2. Применим библиотеки наподобие ramdajs, которые используют Maybe для доступа к значению изнутри и снаружи монады.

3. Предоставим значение по умолчанию, если реальное значение окажется null (т. е. заранее обработаем null-ошибки).

```
//Этап 1. Вместо...
if (user == null) { //не залоггирован
    return indexURLs['en']; //возвращаем страницу по умолчанию
  }

//Используем:
 Maybe(user) //Возвращает Maybe({userObj}) или Maybe(null). То есть данные завёрнуты ВНУТРИ Maybe

//Этап 2. Вместо...
 if (user.prefs.languages.primary && user.prefs.languages.primary != 'undefined') {
    if (indexURLs[user.prefs.languages.primary]) {//если существует локализованная страница,
      return indexURLs[user.prefs.languages.primary];

//Используем:
//библиотеку, умеющую работать с данными внутри Maybe, вроде map.path из Ramda:
 <userMaybe>.map(path(['prefs', 'languages', 'primary']))

//Этап 3. Вместо...
 return indexURLs['en']; //hardcoded default values

//Используем:
//все Maybe-библиотеки предоставляют метод orElse или getOrElse, который вернёт либо актуальные данные, либо значение по умолчанию
<userMayBe>.getOrElse('http://site.com/en')
```

Currying (помогает работать с глобальными данными и мультипараметрическими функциями)

В разделе рассматриваются: чистые функции (Pure functions) и композиция (Composition)

Если мы хотим составить цепочки функций — func1.func2.func3 или (func1(func2(func3())), то каждая из них может брать лишь по одному параметру. Например, если func2 берёт два параметра — func2(param1, param2), то мы не сможем включить её в цепочку!

Но ведь на практике многие функции берут по несколько параметров. Так как же нам их комбинировать? Решение: каррирование (Currying).

Каррирование — это преобразование функции, берущей несколько параметров за один раз, в функцию, берущую только один параметр. Функция не выполнится, пока не будут переданы все параметры.

Кроме того, каррирование можно использовать при обращении к глобальным переменным, т. е. делать это «чисто».

Посмотрим снова на наше решение:

```
//Ниже показано, как работать с глобальными переменными и при этом
делать функции пригодными для включения в цепочки

//Глобал indexURLs сопоставлен с разными языками
let indexURLs = {
    'en': 'http://mysite.com/en',  //English
    'sp': 'http://mysite.com/sp', //Spanish
    'jp': 'http://mysite.com/jp'   //Japanese
}

//Императивное решение
const getUrl = (language) => allUrls[language]; //Просто, но грязно
(обращение к глобальной переменной), и есть риск ошибок

//Функциональное решение
```

```
//До каррирования:
const getUrl = (allUrls, language) => {
    return Maybe(allUrls[language]);
}

//После каррирования:
const getUrl = R.curry(function(allUrls, language) {//curry
преобразует это в функцию с одним аргументом
    return Maybe(allUrls[language]);
});

const maybeGetUrl = getUrl(indexURLs) //Хранит в функции curried
глобальное значение.

//С этого момента maybeGetUrl нужен только один аргумент (язык).
Можно сделать цепочку наподобие
maybe(user).chain(maybeGetUrl).bla.bla
```

Пример 2. Работа с кидающими ошибки функциями и выход немедленно после возникновения ошибки

В разделе рассматривается: монада Either

Монада Maybe очень удобна, если у нас есть значения «по умолчанию» для замены Null-ошибок. Но что насчёт функций, которым действительно нужно кидать ошибки? И как узнать, какая функция кинула ошибку, если мы собрали в цепочку несколько кидающих ошибки функций? То есть нам нужен быстрый отказ (fast-failure).

Например: у нас есть цепочка **func1.func2.func3…**, и если **func2** кинула ошибку, то нужно пропустить **func3** и последующие функции и правильно показать ошибку из **func2** для дальнейшей обработки.

Монада Either

Монады Either отлично подходят для работы с несколькими функциями, когда любая из них может кинуть ошибку и захотеть немедленно после этого выйти, так что мы можем определить, где именно произошла ошибка.

Применение: в приведённом ниже императивном коде мы вычисляем **tax** и **discount** для **item'ов** и отображаем **showTotalPrice**.

Обратите внимание, что функция tax кинет ошибку, если значение цены будет нечисловое. По той же причине кинет ошибку и функция discount. Но, кроме того, discount кинет ошибку, если цена item'а меньше 10.

Поэтому в **showTotalPrice** проводятся проверки на наличие ошибок.

```
//Императивное решение
//Возвращает либо ошибку, либо цену с учётом налога
const tax = (tax, price) => {
  if (!_.isNumber(price)) return new Error("Price must be numeric");

  return price + (tax * price);
};

// Возвращает либо ошибку, либо цену с учётом скидки
const discount = (dis, price) => {
  if (!_.isNumber(price)) return (new Error("Price must be
numeric"));

  if (price < 10) return new Error("discount cant be applied for
items priced below 10");
```

```
  return price - (price * dis);
};

const isError = (e) => e && e.name == 'Error';

const getItemPrice = (item) => item.price;

//Показывает общую цену с учётом налога и скидки. Должен
обрабатывать ошибки.
const showTotalPrice = (item, taxPerc, disount) => {
  let price = getItemPrice(item);
  let result = tax(taxPerc, price);
  if (isError(result)) {
    return console.log('Error: ' + result.message);
  }
  result = discount(discount, result);
  if (isError(result)) {
    return console.log('Error: ' + result.message);
  }
  //показывает результат
  console.log('Total Price: ' + result);
}

let tShirt = { name: 't-shirt', price: 11 };
let pant = { name: 't-shirt', price: '10 dollars' };
let chips = { name: 't-shirt', price: 5 }; //less than 10 dollars
error

showTotalPrice(tShirt) // Сумма: 9,075
showTotalPrice(pant)   // Ошибка: цена должна быть числом
showTotalPrice(chips)  // Ошибка: скидка не применяется к цене ниже
10
```

Посмотрим, как можно улучшить **showTotalPrice** с помощью монады Either и переписать всё в ФП-стиле.

Монада Either предоставляет два конструктора: Either.Left и Either.Right. Их можно считать подклассами Either. **Left и Right — это монады! Идея в том, чтобы хранить ошибки/исключения в Left, а полезные значения — в Right.** То есть в зависимости от

значения создаём экземпляр Either.Left или Either.Right. **Сделав так, мы можем применить к этим значениям map, chain и т. д.**

Хотя и **Left**, и **Right** предоставляют map, chain и пр., конструктор **Left** только хранит ошибки, а все функции реализует конструктор Right, потому что он хранит фактический результат.

Теперь посмотрим, как можно преобразовать наш императивный код в функциональный.

Этап 1. Обернём возвращаемые значения в Left и Right.

Примечание: «обернём» означает «создадим экземпляр какого-то класса». Эти функции внутри себя вызывают new, так что нам не придётся это делать.

```
var Either = require('ramda-fantasy').Either;
var Left = Either.Left;
var Right = Either.Right;

const tax = R.curry((tax, price) => {
  if (!_.isNumber(price)) return Left(new Error("Price must be
numeric")); //<--Обернём ошибку в Either.Left

  return  Right(price + (tax * price)); //<--Обернём результат в
Either.Right
});

const discount = R.curry((dis, price) => {
  if (!_.isNumber(price)) return Left(new Error("Price must be
numeric")); //<--Wrap Error in Either.Left

  if (price < 10) return Left(new Error("discount cant be applied
for items priced below 10")); //<-- Обернём ошибку в Either.Left

  return Right(price - (price * dis)); //<--Обернём результат в
```

```
Either.Right
});
```

Этап 2. Обернём исходное значение в **Right**, потому что оно валидное и мы можем его комбинировать (compose).

```
const getItemPrice = (item) => Right(item.price);
```

Этап 3. Создадим две функции: одну для обработки ошибки, вторую для обработки результата. Обернём их в **Either.either** (из ramda-fantasy.js api).

Either.either берёт три параметра: обработчика результата, обработчика ошибки и монаду Either. Either каррирована, поэтому мы можем передать обработчики сейчас, а Either (третий параметр) — позже.

Как только Either.either получает все три параметра, она передаёт Either либо в обработчик результата, либо в обработчик ошибки, в зависимости от того, чем является Either — Right или Left.

```
const displayTotal = (total) => { console.log('Total Price: ' +
total) };
const logError = (error) => { console.log('Error: ' +
error.message); };
const eitherLogOrShow = Either.either(logError, displayTotal);
```

Этап 4. Используем метод **chain** для комбинирования функций, кидающих ошибки. Передадим их результаты в Either.either (eitherLogOrShow), которая позаботится о том, чтобы ретранслировать их в обработчик результата или ошибки.

```
const showTotalPrice = (item) =>
eitherLogOrShow(getItemPrice(item).chain(apply25PercDisc).chain(addC
aliTax));
```

Соберём всё вместе:

```
const tax = R.curry((tax, price) => {
  if (!_.isNumber(price)) return Left(new Error("Price must be
numeric"));

  return  Right(price + (tax * price));
});

const discount = R.curry((dis, price) => {
  if (!_.isNumber(price)) return Left(new Error("Price must be
numeric"));

  if (price < 10) return Left(new Error("discount cant be applied
for items priced below 10"));

  return Right(price - (price * dis));
});

const addCaliTax = (tax(0.1));//налог 10 %

const apply25PercDisc = (discount(0.25));// скидка 25 %

const getItemPrice = (item) => Right(item.price);

const displayTotal = (total) => { console.log('Total Price: ' +
total) };

const logError = (error) => { console.log('Error: ' +
error.message); };

const eitherLogOrShow = Either.either(logError, displayTotal);

//api
const showTotalPrice = (item) =>
eitherLogOrShow(getItemPrice(item).chain(apply25PercDisc).chain(addC
aliTax));

let tShirt = { name: 't-shirt', price: 11 };
```

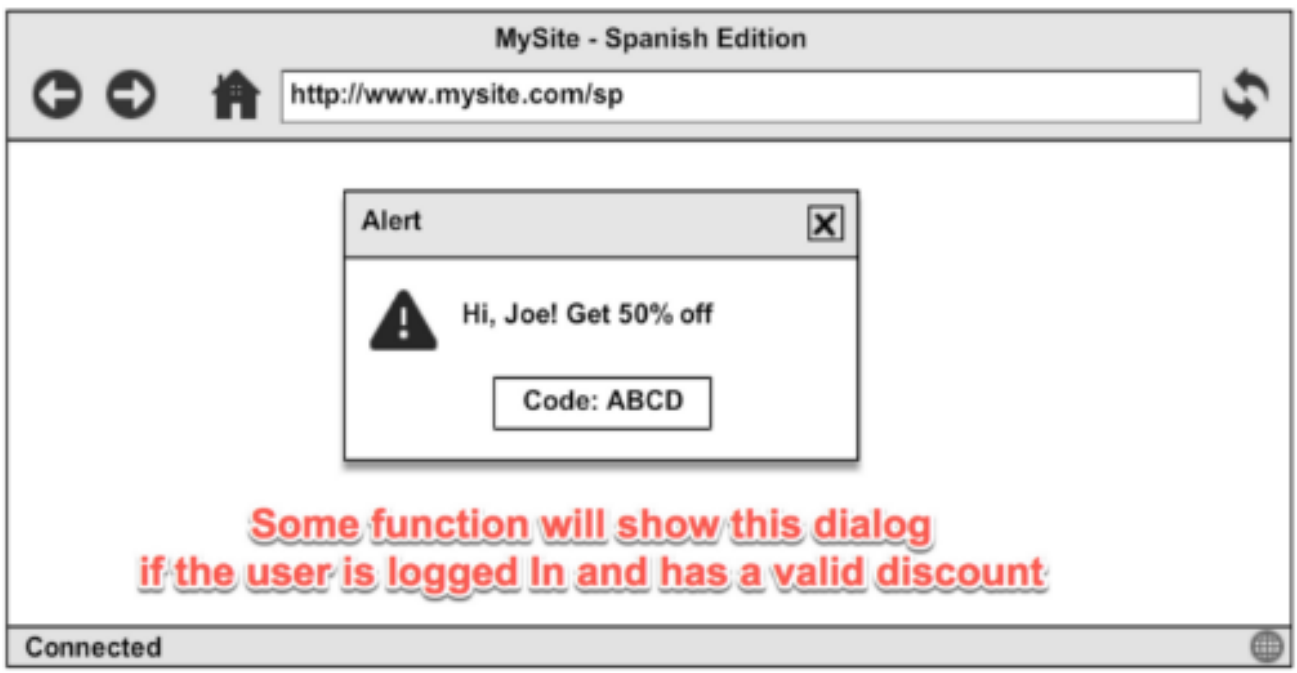
```
let pant = { name: 't-shirt', price: '10 dollars' }; //error
let chips = { name: 't-shirt', price: 5 }; //less than 10 dollars
error

showTotalPrice(tShirt) // Сумма: 9,075
showTotalPrice(pant)   // Ошибка: цена должна быть числом
showTotalPrice(chips)  // Ошибка: скидка не применяется к цене ниже
10
```

Пример 3. Присвоение значения потенциальным Null-объектам

***Использована ФП-концепция: аппликативность (Applicative)

Применение: допустим, нужно дать пользователю скидку, если он залогинился и если действует промоакция (т. е. существует скидка).



Воспользуемся методом **applyDiscount**. Он может кидать null-ошибки, если пользователь (слева) или скидка (справа) является null.

```
//Добавим скидку в объект user, если существует и пользователь, и скидка.
//Null-ошибки кидаются в том случае, если пользователь или скидка является null.
const applyDiscount = (user, discount) => {
    let userClone = clone(user);// С помощью какой-нибудь библиотеки сделаем копию
    userClone.discount = discount.code;
   return userClone;
}
```

Посмотрим, как можно решить это с помощью аппликативности.

Аппликативность (Applicative)

Любой класс, имеющий метод ap и реализующий спецификацию Applicative, называется аппликативным. Такие классы могут использоваться в функциях, которые работают с null-значениями как с левой стороны (пользователь) уравнения, так и с правой (скидка).

Монады Maybe (как и все монады) тоже реализуют спецификацию ap, а значит, они тоже аппликативные, а не просто монады. Поэтому на функциональном уровне мы можем использовать монады Maybe для работы с null. Посмотрим, как заставить работать applyDiscount с помощью монады Maybe, используемой как аппликативная.

Этап 1. Обернём потенциальные null-значения в монады Maybe.

```
const maybeUser = Maybe(user);
const maybeDiscount = Maybe(discount);
```

Этап 2. Перепишем функцию и каррируем её, чтобы передавать по одному параметру за раз.

```
// перепишем функцию и каррируем её, чтобы
// передавать по одному параметру за раз
var applyDiscount = curry(function(user, discount) {
       user.discount = discount.code;
       return user;
});
```

Этап 3. Передадим через map первый аргумент (maybeUser) в applyDiscount.

```
// передадим через map первый аргумент (maybeUser) в applyDiscount
const maybeApplyDiscountFunc = maybeUser.map(applyDiscount);
//Обратите внимание, что поскольку applyDiscount каррирована и map
передаст только один параметр, то возвращаемый результат
(maybeApplyDiscountFunc) будет обёрнутой в монаду Maybe функцией
applyDiscount, которая в своём замыкании теперь содержит
maybeUser(первый параметр).
```

Иными словами, теперь у нас есть функция, обёрнутая в монаду!

Этап 4. Работаем с maybeApplyDiscountFunc.

На этом этапе maybeApplyDiscountFunc может быть:

1) функцией, обёрнутой в Maybe, — если пользователь существует;
2) Nothing (подклассом Maybe) — если пользователь не существует.

Если пользователь не существует, то возвращается Nothing, а все последующие действия с ним полностью игнорируются. Так что если мы передадим второй аргумент, то ничего не произойдёт. Также не будет null-ошибок.

Если пользователь существует, то можем попытаться передать второй аргумент в maybeApplyDiscountFunc через map, чтобы выполнить функцию:

```
maybeDiscount.map(maybeApplyDiscountFunc)! // ПРОБЛЕМА!
```

Ой-ёй: map не знает, как выполнять функцию (maybeApplyDiscountFunc), когда она сама внутри Maybe!

Поэтому нам нужен другой интерфейс для работы по такому сценарию. И этот интерфейс — ap!

Этап 5. Освежим информацию о функции ap. Метод ap берёт другую монаду Maybe и передаёт/применяет к ней хранимую им в данный момент функцию.

```
class Maybe {
  constructor(val) {
    this.val = val;
  }
  ...
  ...
  //ap берёт другую maybe и применяет к ней хранящуюся в нём
функцию.
  //this.val ДОЛЖЕН быть функцией или Nothing (и не может быть
строкой или числом)
  ap(differentMayBe) {
     return differentMayBe.map(this.val);
  }
}
```

Можем просто применить (ap) maybeApplyDiscountFunc к maybeDiscount вместо использования map, как показано ниже. И это будет прекрасно работать!

```
maybeApplyDiscountFunc.ap(maybeDiscount)
//Поскольку applyDiscount хранится внутри this.val в обёртке
maybeApplyDiscountFunc:
maybeDiscount.map(applyDiscount)
//Далее, если maybeDiscount имеет скидку, то функция выполняется.
Если maybeDiscount является Null, то ничего не происходит.
```

Для сведения: очевидно, в спецификации Fantasy Land внесли изменение. В старой версии нужно было писать: Just(f).ap(Just(x)), где f — это функция, x — значение. В новой версии нужно писать Just(x).ap(Just(f)). Но реализации по большей части пока не изменились. Спасибо keithalexander.

Подведём итог. Если у вас есть функция, работающая с несколькими параметрами, каждый из которых может быть null, то сначала каррируйте её, а затем поместите внутрь Maybe. Также

поместите в Maybe все параметры, а для исполнения функции воспользуйтесь ap.

Функция curryN

Мы уже знакомы с каррированием. Это простое преобразование функции, чтобы она брала не несколько аргументов сразу, а по одному.

```
//Пример каррирования:
const add = (a, b) =>a+b;
const curriedAdd = R.curry(add);
const add10 = curriedAdd(10);//Передаёт первый аргумент. Возвращает функцию, берущую второй (b) параметр.
//Запускает функцию после передачи второго параметра.
add10(2) // -> 12 //внутренне запускает add с 10 и 2.
```

Но что если вместо добавления всего двух чисел функция **add** могла суммировать все числа, передаваемые в неё в качестве аргументов?

```
const add = (...args) => R.sum(args); //Суммирует все числа в аргументах
```

Мы всё ещё можем каррировать её, ограничив количество аргументов с помощью **curryN**:

```
//пример curryN
const add = (...args) => R.sum(args);
//пример CurryN:
const add = (...args) => R.sum(args);
const add3Numbers = R.curryN(3, add);
const add5Numbers = R.curryN(5, add);
```

```
const add10Numbers = R.curryN(10, add);
add3Numbers(1,2,3) // 6
add3Numbers(1) // возвращает функцию, берущую ещё два параметра.
add3Numbers(1, 2) // возвращает функцию, берущую ещё один параметр.
```

Использование curryN для ожидания количества вызовов функции

Допустим, нам нужна функция, которая пишет в лог только тогда, когда мы вызвали её три раза (один и два вызова игнорируются). Например:

```
//грязное решение
let counter = 0;
const logAfter3Calls = () => {
 if(++counter == 3)
   console.log('called me 3 times');
}
logAfter3Calls() // Ничего не происходит
logAfter3Calls() // Ничего не происходит
logAfter3Calls() // 'called me 3 times'
```

Можем эмулировать такое поведение с помощью curryN.

```
//чистое решение
const log = () => {
   console.log('called me 3 times');
}
const logAfter3Calls = R.curryN(3, log);
//вызов
logAfter3Calls('')('')('')//'called me 3 times'
//Примечание: мы передаём '', чтобы удовлетворить CurryN фактом
передачи параметра.
```

Примечание: мы будем использовать эту методику при аппликативной валидации.

Пример 4. Сбор и отображение ошибок

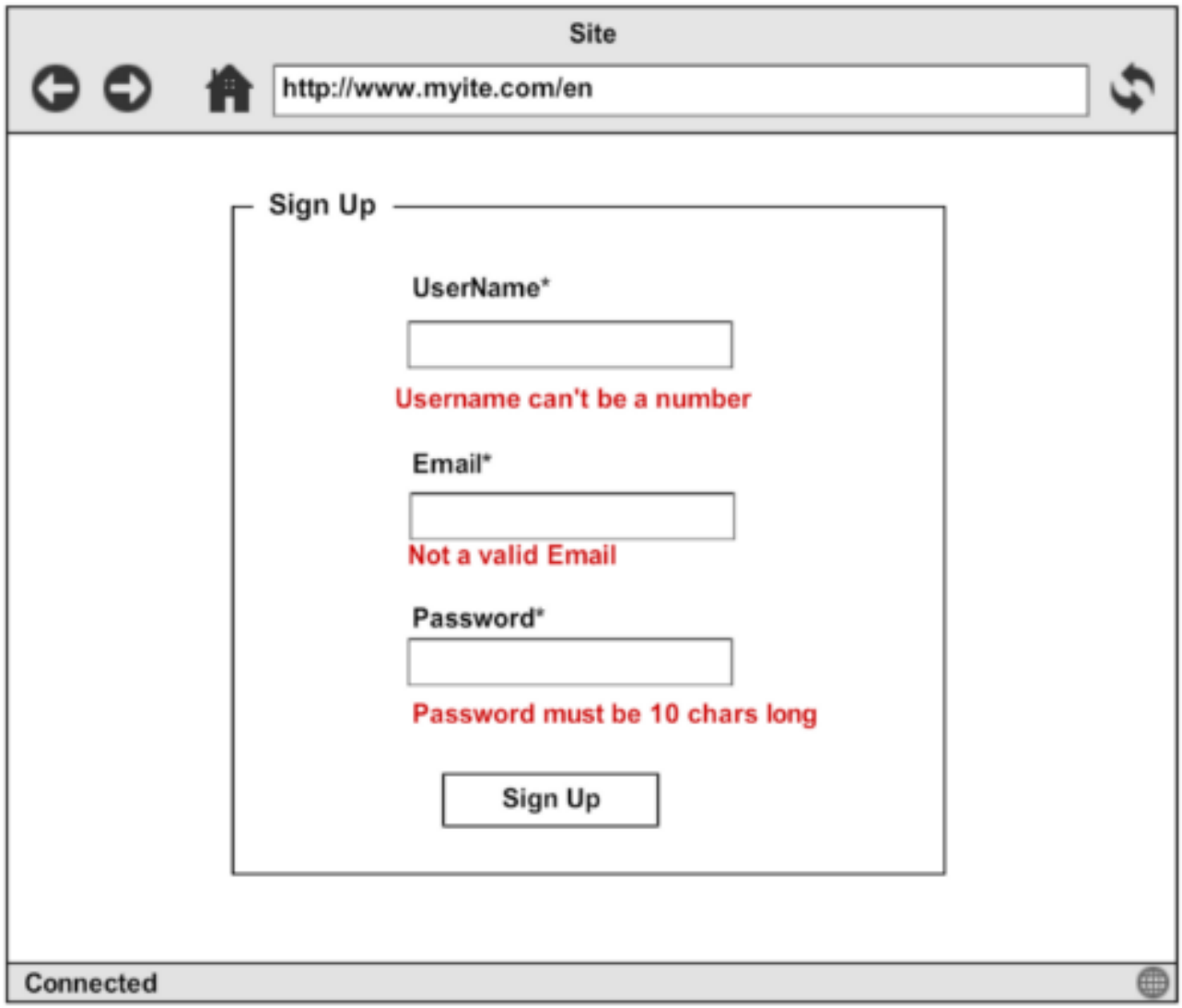
В разделе рассматривается: валидация (известна как функтор Validation, аппликативность Validation, монада Validation).

Валидациями обычно называют **аппликативность Validation (Validation Applicative)**, потому что она чаще всего применяется для валидации с использованием функции ap (apply).

Валидации аналогичны **монадам Either** и часто используются для работы с комбинациями функций, кидающих ошибки. Но в отличие от Either, в которых мы для комбинирования обычно применяем метод chain, в монадах Validation мы для этого обычно используем метод ap. Кроме того, если chain позволяет собирать только первую ошибку, то **ap, особенно в монадах Validation, даёт собирать в массив все ошибки.**

Обычно эти монады используются при валидации форм ввода, когда нам нужно сразу показать все ошибки.

Применение: у нас есть форма регистрации, которая проверяет имя, пароль и почту с помощью трёх функций (isUsernameValid, isPwdLengthCorrect и ieEmailValid). Нам нужно одновременно показать все три ошибки, если они возникнут.



Для этого воспользуемся функтором Validation. Посмотрим, как это можно реализовать с помощью аппликативности Validation.

Возьмём из folktalejs библиотеку data.validation, в ramda-fantasy она ещё не реализована. По аналогии с монадой Either у нас есть два конструктора: **Success** и **Failure**. Это подклассы, каждый из которых реализует спецификации Either.

Этап 1. Для использования валидации нам нужно обернуть правильные значения и ошибки в конструкторы **Success** и **Failure**

(т. е. создать экземпляры этих классов).

```
const Validation = require('data.validation') //из folktalejs
const Success = Validation.Success
const Failure = Validation.Failure
const R = require('ramda');
//вместо:
function isUsernameValid(a) {
    return /^(0|[1-9][0-9]*)$/.test(a) ?
           ["Username can't be a number"] : a
}
//используем:
function isUsernameValid(a) {
    return /^(0|[1-9][0-9]*)$/.test(a) ?
         Failure(["Username can't be a number"]) : Success(a)
}
```

Повторите процесс для всех проверочных функций, кидающих ошибки.

Этап 2. Создадим пустую функцию (dummy function) для хранения успешного статуса проверки.

```
const returnSuccess = () => 'success';//просто
возвращает success
```

Этап 3. Используем curryN для многократного применения ap

С ap есть проблема: левая часть должна быть функтором (или монадой), содержащим **функцию**.

Допустим, нам нужно многократно применить ap. Это будет работать, только если monad1 содержит функцию. И результат

monad1.ap(monad2), т. е. **resultingMonad**, тоже должен быть монадой с функцией, чтобы можно было применить ap к monad3.

```
let finalResult = monad1.ap(monad2).ap(monad3)
//можно переписать как
let resultingMonad = monad1.ap(monad2)
let finalResult = resultingMonad.ap(monad3)
//будет работать, если monad1 содержит функцию, а результат
monad1.ap(monad2) является другой монадой (resultingMonad) с
функцией
```

В общем, чтобы дважды применить ap, нам нужны две монады, содержащие функции.

В данном случае у нас три функции, к которым нужно применить ap.

Допустим, мы сделали что-то такое.

```
         Success(returnSuccess)
        .ap(isUsernameValid(username)) //работает
        .ap(isPwdLengthCorrect(pwd))//не работает
        .ap(ieEmailValid(email))//не работает
```

Предыдущий код не станет работать, потому что результатом Success(returnSuccess).ap(isUsernameValid(username)) будет значение. И мы не сможем применить ap ко второй и третьей функциям.

Можно использовать curryN, чтобы возвращать функцию до тех пор, пока она не будет вызвана N раз.

```
//3 – потому что вызываем ap три раза.
let success = R.curryN(3, returnSuccess);
```

Теперь каррированная **success возвращает функцию три раза**.

```
function validateForm(username, pwd, email) {
    //3 – потому что вызываем ap три раза.
    let success = R.curryN(3, returnSuccess);
    return Success(success)// по умолчанию; используется для трёх ap
        .ap(isUsernameValid(username))
        .ap(isPwdLengthCorrect(pwd))
        .ap(ieEmailValid(email))
}
```

Соберём всё вместе:

```
const Validation = require('data.validation') //из folktalejs
const Success = Validation.Success
const Failure = Validation.Failure
const R = require('ramda');

function isUsernameValid(a) {
    return /^(0|[1-9][0-9]*)$/.test(a) ? Failure(["Username can't be
a number"]) : Success(a)
}

function isPwdLengthCorrect(a) {
    return a.length == 10 ? Success(a) : Failure(["Password must be
10 characters"])
}

function ieEmailValid(a) {
    var re = /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|
(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-
zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;

    return re.test(a) ? Success(a) : Failure(["Email is not valid"])
}

const returnSuccess = () => 'success';//просто возвращает success
```

```
function validateForm(username, pwd, email) {
    let success = R.curryN(3, returnSuccess);// 3 – потому что
вызываем ap три раза.
    return Success(success)
        .ap(isUsernameValid(username))
        .ap(isPwdLengthCorrect(pwd))
        .ap(ieEmailValid(email))
}

validateForm('raja', 'pwd1234567890', 'r@r.com').value;
//Output: success

validateForm('raja', 'pwd', 'r@r.com').value;
//Output: ['Password must be 10 characters' ]

validateForm('raja', 'pwd', 'notAnEmail').value;
//Output: ['Password must be 10 characters', 'Email is not valid']

validateForm('123', 'pwd', 'notAnEmail').value;
//['Username can\'t be a number', 'Password must be 10 characters',
'Email is not valid']
```


Похожие публикации

ПЕРЕВОД

AloneCoder • 5 октября 2017 в 18:44

Руководство для начинающих по прогрессивным веб-приложениям и фронтенду

130

ПЕРЕВОД

AloneCoder • 1 августа 2017 в 20:04

Понимание событийной архитектуры Node.js

9

ПЕРЕВОД

AloneCoder • 6 июня 2017 в 16:38

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов

20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE

24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить 85

Pochtoycom • вчера в 13:06

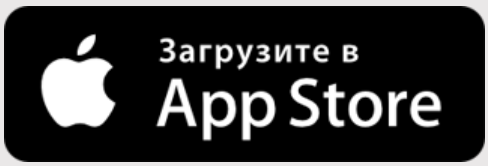
Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе 140

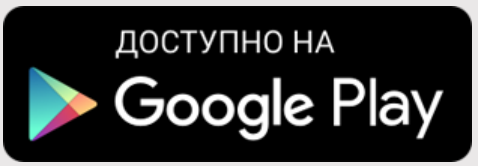
marks • вчера в 14:19

Дела шпионские (часть 1) 16

TashaFridrih • вчера в 13:16

Мобильное приложение





Полная версия