

Основы функционального программирования

Сергей Зинчук

О чем будем говорить

- Функции высшего порядка
- Замыкания
- Частичное применение
- Каррирование
- Сопоставление с образцом

И не только :)

Функции в функциональных языках

- *Объекты первого класса (first-class object)*
- *Чистые (pure functions)*
- *Возвращают на одинаковых данных один и тот же результат, независимо от места вызова (Referential transparency)*
- *Могут быть ленивыми (lazy evaluation)*
- **Имеют тип**

Детальнее смотрим в [2]

Основа...о которой многие слышали – ФВП
Функция высшего порядка(Higher-order function)

Функция является ФВП если:

- Хотя бы один из ее аргументов – это функция
- Возвращаемое значение – функция

Как это выглядит ?

ФВП – Java-пример

```
public interface Func<R,P> {  
    public R apply(P param);  
}  
  
public class TimesTwo implements Func<Double, Double> {  
    @Override  
    public Double apply(Double param) {  
        return param*2;  
    }  
}
```

ФВП – Java-пример

```
public class Demo {  
    public static void printResult(Func<Double, Double> f, Double  
    param) {  
        System.out.println(f.apply(param));  
    }  
    public static void main(String[] args) {  
        double param = 5;  
        TimesTwo tt = new TimesTwo();  
        printResult(tt, param); //наконец вывели в консоль 10  
    }  
}
```

ФВП – пример на Scala

// Более кратко и лаконично

```
object Hof extends App {
```

```
  def TimesTwo(param: Double): Double = param * 2
```

```
  def printValue(f: Double => Double, param: Double) = println(f(param))
```

Тип параметра – функция из Double в Double

```
  printValue(TimesTwo, 5) //10
```

```
  printValue(x => x*2, 5) // и снова 10, но 1й параметр – анонимная  
                        // функция
```

```
}
```

ФВП – пример на Scala №2

```
val people = Array("Андрей", "Сергей", "Александр",  
"Владимир")
```

```
people.foreach { name:String => println("Person: " + name) }
```

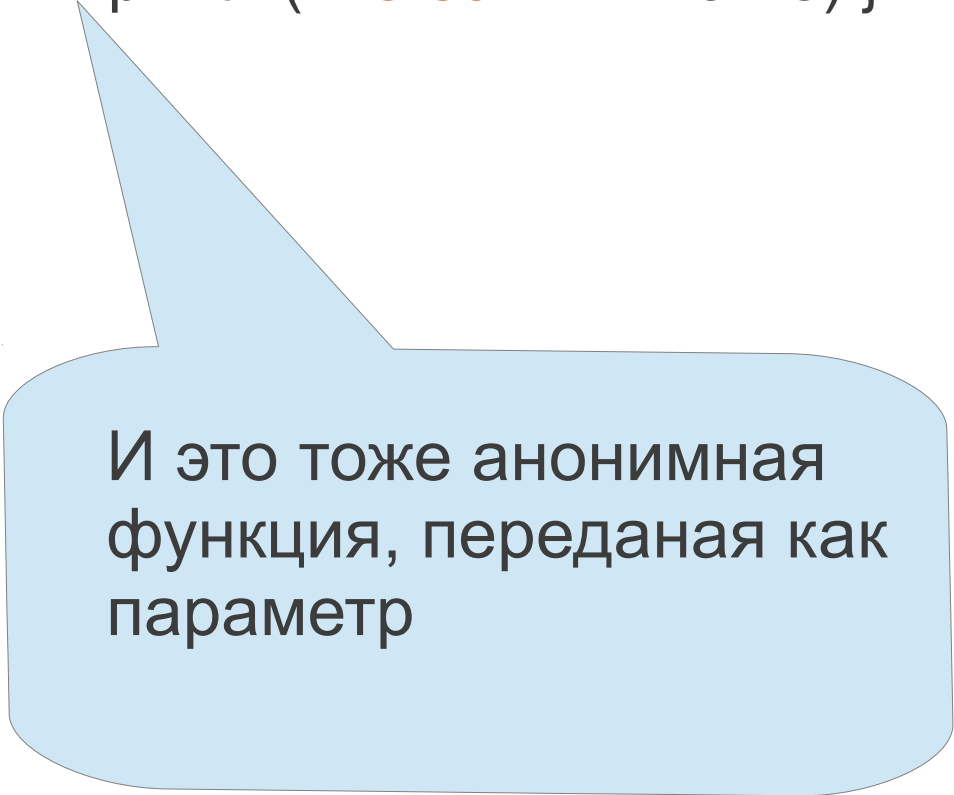
Prints out:

Person: Андрей

Person: Сергей

Person: Александр

Person: Владимир



И это тоже анонимная функция, переданная как параметр

Замыкания (Closures)

Замыкание — процедура или функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции и не в качестве её параметров (а в окружающем коде). (Wiki)

- Без замыканий проблематично написать нетривиальную ФВП
- Поддержка замыканий в рамках некоторого языка связана с “проблемой функционального аргумента”(The Funarg problem)

По сути, замыкание – это функция, использующая переменные из области видимости, в которой была создана.

Замыкания – подводящий пример

```
scala> (x: Int) => x + inc
```

```
<console>:8: error: not found: value inc
```

```
(x: Int) => x + inc
```

```
^
```

На какое именно значение нам нужно увеличить x ???

Замечание: связанные и свободные переменные

С точки зрения функционального литерала (*function literal*)

`(x: Int) => x + inc`

переменная `inc` – **свободная переменная** (*free variable*) ,
поскольку она используется, но **не определена** внутри
литерала;

переменная `x` – **связанная переменная** (*bound variable*) ,
поскольку определена как единственный параметр функции

Связанные и свободные переменные

// Добавим необходимую информацию во внешний контекст

```
scala> var inc = 1
```

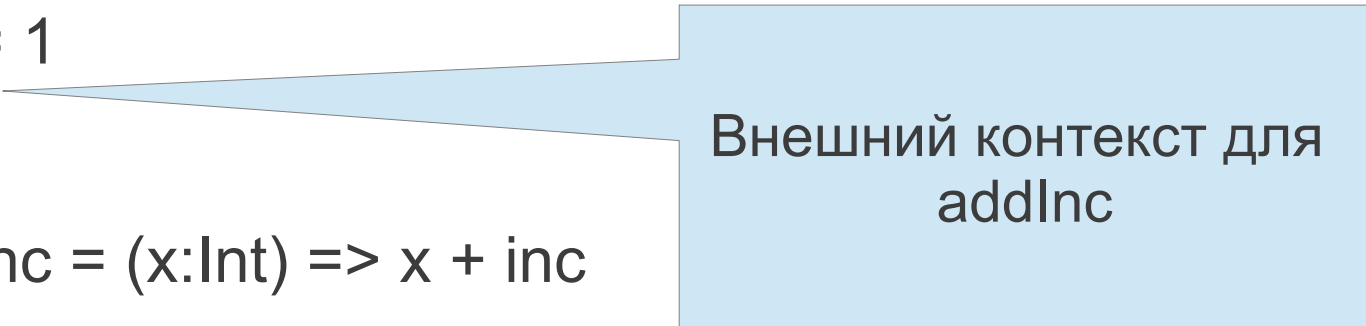
```
inc: Int = 1
```

```
scala> val addInc = (x:Int) => x + inc
```

```
addInc: Int => Int = <function1>
```

```
scala> addInc(10)
```

```
res1: Int = 11
```



Внешний контекст для
addInc

Что такое addInc ???

Связанные и свободные переменные

addInc – это функциональный объект, полученный вследствие “закрытия” (“closing”) функционального литерала с помощью связывания свободных переменных, т. е.

ЗАМЫКАНИЕ

Особенность замыкания

Замыкание хранит **ссылки** на внешние переменные, но не их значения => оно способно “отслеживать” изменение значений этих переменных

```
scala> inc = 9999
```

```
inc: Int = 9999
```

```
scala> addInc(10)
```

```
res3: Int = 10009
```

Пример замыкания - инкрементор

```
def makeIncreaser(inc: Int): (Int => Int) = {  
  def concreteIncreaser(x: Int): Int = {  
    x + inc  
  } concreteIncreaser // вернули инкрементор,  
  увеличивающий x на конкретное значение inc  
}  
  
val incToTen = makeIncreaser(10);  
println(incToTen(5)) //15
```

Функция—
инкрементор на 10,
созданная после
связывания inc со
значением 10

Пример замыкания - инкрементор

// Более кратко можно написать так

```
def makeIncreaser(inc: Int): (Int => Int) = (x: Int) => x + inc
```

```
val five = 5
```

```
val twelve = 12
```

```
val incByFive = makeIncreaser(five)
```

```
val incByTwelve = makeIncreaser(twelve)
```

```
println(incByFive(10)) //15
```

```
println(incByTwelve(10)) //22
```


Каррирование и частичное применение

“Частичным применением n -арной функции f называется конструкция, значением которой является $(n-k)$ -арная функция, соответствующая f при фиксированных значениях некоторых k из n аргументов.” [1]

Частичное применение - пример

```
def drawLine (width: Double, color: Color, style: Linestyle,  
a:Point, b: Point): Unit = {  
    //как-то рисуем...  
}
```

```
val drawSolidLine = drawLine(_:Double, _:Color,  
    new Linestyle("SOLID"), _:Point, _:Point)
```

```
drawSolidLine(2.5, new Color("Black"),new Point(2,4), new  
Point(3,1))
```

Каррирование

Техника преобразования *n-арной* функции так, что она может быть вызвана как цепочка из *n* функций, каждая из которых принимает по одному аргументу.

```
scala> def plainOldSum(x: Int, y: Int) = x + y
```

```
plainOldSum: (x: Int,y: Int)Int
```

```
scala> plainOldSum(1, 2)
```

```
res4: Int = 3
```

```
scala> def curriedSum(x: Int)(y: Int) = x + y
```

```
curriedSum: (x: Int)(y: Int)Int
```

```
scala> curriedSum(1)(2)
```

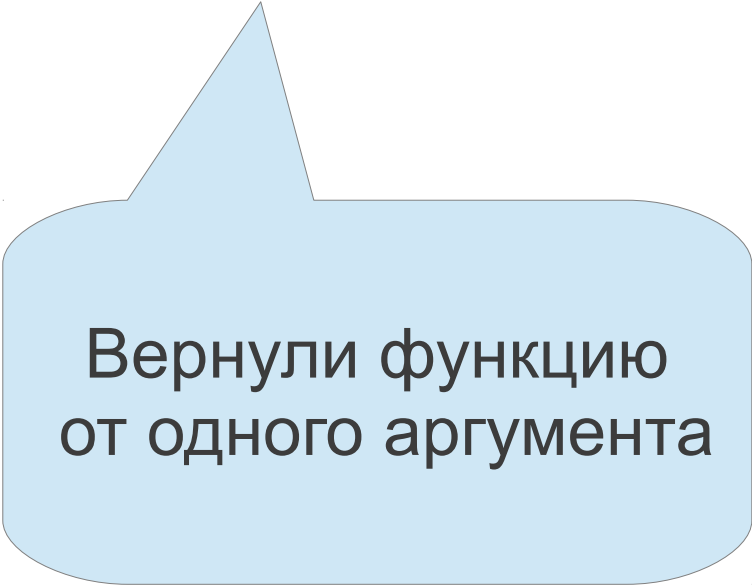
Замечание: асоциативность операторов

В лямбда-исчислении и в Scala оператор применения функции $()$ *левоассоциативен*, т.е

`curriedSum(1)(2)`

эквивалентно

`(curriedSum(1)) (2)`



Вернули функцию
от одного аргумента

Каррирование как частичное применение

“Частный случай частичного применения, при котором фиксируется несколько *первых* аргументов функции.”[1]

```
def curriedDrawLine (width:Double)(color: Color)(style: Linestyle) ( a:Point)(b:Point): Unit = {
```

```
//Как-то нарисовали
```

```
}
```

```
val drawThinBlackLine = curriedDrawLine(1.0)(new Color("Black"))  
(_:Linestyle)(_:Point)(_:Point)
```

```
drawThinBlackLine(new Linestyle("SOLID"), new Point(2,3), new Point(5,6))
```

Суть карринга – специализация ПОД ТИПЫ ДАННЫХ

//Сума значений функции на интервале

```
def sum(f: Int => Int)(a: Int, b: Int): Int = {  
  if (a > b) 0 else f(a) + sum(f) (a + 1, b)  
}
```

```
def cube(x: Int) = x * x * x
```

```
def square(x: Int) = x*x
```

```
sum(cube)(1,5) // 225
```

```
sum(square)(1,5) // 55
```

Алгебраический тип данных (Algebraic Data Type)

“Тип данных, состоящий из нескольких различных разновидностей (возможно, составных) термов (значений)”
[1]

```
abstract class Shape
```

```
case class Rectangle(topLeft: Point, bottomRight: Point)  
extends Shape
```

```
case class Circle(center: Point, radius: Double) extends  
Shape
```

```
case class Triangle(a: Point, b: Point, c: Point) extends Shape
```

Рекурсивный ADT

```
abstract class Expr
```

```
case class Var(name: String) extends Expr
```

```
case class Number(num: Double) extends Expr
```

```
case class UnOp(operator: String, arg: Expr) extends  
Expr
```

```
case class BinOp(operator: String, left: Expr, right: Expr)  
extends Expr
```


Сопоставление с образцом (Pattern matching)

//Псевдокод

case EXPRESSION of

PATTERN1 → VALUE1

PATTERN2 → VALUE2

...

EXPRESSION — произвольное выражение, обладающее значением (обычно принадлежащим к алгебраическому типу)

VALUE — выражения или операторы (statement)

PATTERN — собственно образцы

Замечание об образцах

“Образец — это описание «формы» ожидаемой структуры данных: образец сам по себе похож на литерал структуры данных (т. е. он состоит из конструкторов алгебраических типов и литералов примитивных типов: целых, строковых и т. п.), однако может содержать метапеременные — «дырки», обозначающие: «значение, которое встретится в данном месте, назовем данным именем».”[1]

Обработка ADT через Pattern matching

```
def simplifyTop(expr: Expr): Expr = expr match {  
  case UnOp("-", UnOp("-", e)) => e // Double negation  
  case BinOp("+", e, Number(0)) => e // Adding zero  
  case BinOp("*", e, Number(1)) => e // Multiplying by one  
  case _ => expr  
}  
  
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))  
res4: Expr = Var(x)
```

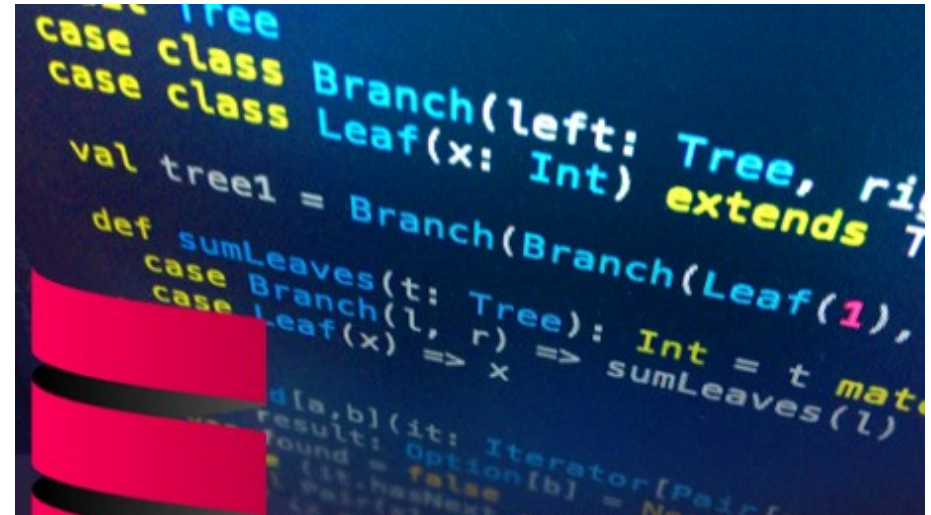
О чем мы не говорили

- Проблема вывода типов в функциональных языках ([Type Inference](#))
- Чисто функциональные структуры данных ([Purely functional data structures](#))
- Свертка ([Fold](#))
- Классы типов ([Type Classes](#))

Functional Programming Principles in Scala

Martin Odersky

Learn about functional programming, and how it can be effectively combined with object-oriented programming. Gain practice in writing clean functional code, using the Scala programming language.



```
case class Branch(left: Tree, right: Tree) extends Tree
case class Leaf(x: Int) extends Tree
val tree1 = Branch(Branch(Leaf(1), Leaf(2)), Leaf(3))
def sumLeaves(t: Tree): Int = t match {
  case Branch(l, r) => sumLeaves(l) + sumLeaves(r)
  case Leaf(x) => x
}
def range(a, b)(it: Iterator[Pair]) = {
  result: Option[Pair] = None
  found = false
  while (it.hasNext) {
    pair = it.next()
    if (pair.first == a && pair.second == b) {
      result = Some(pair)
      found = true
    }
  }
  result
}
```

<https://www.coursera.org/course/progfun>

Next Session: Mar 25th 2013

Works consulted

1. *Евгений Кирпичев*. Элементы функциональных языков. Практика функционального программирования. Выпуск 3
2. *Роман Душкин*. Функции и функциональный подход. Практика функционального программирования. Выпуск 1
3. *Влад Патрышев*. Почему Скала. Практика функционального программирования. Выпуск 6
4. *Martin Odersky, Lex Spoon, Bill Venners*. Programming in Scala, 2nd edition. Artima, 2011.
5. *Dean Wampler, Alex Payne*. Programming Scala. O'Reilly, 2009