Лекция №8. Функциональная парадигма программирования. Введение в язык Scala.

Функциональный подход к программированию
Преимущества языка Scala
Элементы программирования на языке Scala
Выполнение вычислений
Другие замечания

Парадигмы программирования

- императивное программирование
- функциональное программирование
- логическое программирование
- объектно-ориентированное программирование

Императивное программирование

Включает:

- изменение значений переменных: c++

- присваивание: c := c + d

- управляющие конструкции: if - then - else, while, break, continue, return и т.д.

Существует проблема масштабирования

Функциональное программирование (теория)

Математическая теория включает:

- один и более типов данных

- операции над этими типами

- законы, описывающие отношения между значениями и операциями

и не включает модификаций каких-либо значений

Обзор функциональных языков и история ФП

- Программирование без переменных, присваиваний, циклов и других императивных конструкций
  - Lisp (чистый) (1959)
  - XSLT (1999), XPath, XQuery (2003)
  - FP (1977)
  - Haskell (без I/O Monad, UsafePerformIO) (1970)

- Программирование, основанное на использовании функций (функции могут быть входными и выходными значениями, частью композиций)
  - Lisp, Scheme (1976), Racket (1994), Clojure (2007)
  - SML (1986), OCaml (2000), F# (2005)
  - Haskell
  - Scala (2003)
  - Smalltalk (1978), Ruby (1995)

“Scala лучше Java в некоторых аспектах. Перед тем как начинать изучать язык Scala, очистите свой разум, из этого выйдет больше толку.”

Scala Школа!
https://twitter.github.io/scala_school/ru/index.html

Совместимость с Java

Author.scala

```scala
class Author(val firstName: String,
    val lastName: String) extends Comparable[Author] {

  override def compareTo(that: Author) = {
    val lastNameComp = this.lastName compareTo that.lastName
    if (lastNameComp != 0) lastNameComp
    else this.firstName compareTo that.firstName
  }
}

object Author {
  def loadAuthorsFromFile(file: java.io.File): List[Author] =
}
```

App.java

```java
import static scala.collection.JavaConversions.asJavaCollection;

public class App {
    public List<Author> loadAuthorsFromFile(File file) {
        return new ArrayList<Author>(asJavaCollection(
            Author.loadAuthorsFromFile(file)));
    }

    public void sortAuthors(List<Author> authors) {
        Collections.sort(authors);
    }

    public void displaySortedAuthors(File file) {
        List<Author> authors = loadAuthorsFromFile(file);
        sortAuthors(authors);
        for (Author author : authors) {
            System.out.println(
                author.lastName() + ", " + author.firstName());
        }
    }
}
```

Вывод типов

Type inference

```
scala> class Person(val name: String, val age: Int) {
     |   override def toString = s"$name ($age)"
     | }
defined class Person

scala> def underagePeopleNames(persons: List[Person]) =
     |   for (person <- persons; if person.age < 18)
     |     yield person.name
     | }
underagePeopleNames: (persons: List[Person])List[String

scala> def createRandomPeople() = {
     |   val names = List("Alice", "Bob", "Carol",
     |       "Dave", "Eve", "Frank")
     |   for (name <- names) yield {
     |     val age = (Random.nextGaussian()*8 + 20).toI
     |     new Person(name, age)
     |   }
     | }
createRandomPeople: ()List[Person]

scala> val people = createRandomPeople()
people: List[Person] = List(Alice (16), Bob (16), Carol

scala> underagePeopleNames(people)
res1: List[String] = List(Alice, Bob, Frank)
```

Конкурентные / распределённые вычисления

Concurrent/Distributed

```
1.  val x = future { someExpensiveComputation() }
2.  val y = future { someOtherExpensiveComputation() }
3.  val z = for (a <- x; b <- y) yield a*b
4.  for (c <- z) println("Result: " + c)
5.  println("Meanwhile, the main thread goes on!")
```

Трейты

Traits

```
1.  abstract class Spacecraft {
2.    def engage(): Unit
3.  }
4.  trait CommandoBridge extends Spacecraft {
5.    def engage(): Unit = {
6.      for (_ <- 1 to 3)
7.        speedUp()
8.    }
9.    def speedUp(): Unit
10. }
11. trait PulseEngine extends Spacecraft {
12.   val maxPulse: Int
13.   var currentPulse: Int = 0
14.   def speedUp(): Unit = {
15.     if (currentPulse < maxPulse)
16.       currentPulse += 1
17.   }
18. }
19. class StarCruiser extends Spacecraft
20.                       with CommandoBridge
21.                       with PulseEngine {
22.   val maxPulse = 200
23. }
```

Сопоставление с образцом

Pattern matching

```
1.  // Define a set of case classes for representing binary trees.
2.  sealed abstract class Tree
3.  case class Node(elem: Int, left: Tree, right: Tree) extends Tr
4.  case object Leaf extends Tree
5.
6.  // Return the in-order traversal sequence of a given tree.
7.  def inOrder(t: Tree): List[Int] = t match {
8.    case Node(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
9.    case Leaf          => List()
10. }
```

Функции высших порядков

Scala

```scala
val people: Array[Person]

// Partition `people` into two arrays `minors` and `ad
// Use the higher-order function `(_.age < 18)` as a p
val (minors, adults) = people partition (_.age < 18)
```

Java

```java
List<Person> people;

List<Person> minors = new ArrayList<Person>(people.siz
List<Person> adults = new ArrayList<Person>(people.siz
for (Person person : people) {
    if (person.getAge() < 18)
        minors.add(person);
    else
        adults.add(person);
}
```

Сравнение с Java: Объявление класса

```java
public class Person {
  public final String name;
  public final int age;
  Person(String name, int age) {
      this.name = name;
      this.age = age;
  }
}
```

```
class Person(val name: String,
             val age: Int)
```

Сравнение с Java: Использование класса

```
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{  ArrayList<Person> minorsList = new ArrayList<Person>();
   ArrayList<Person> adultsList = new ArrayList<Person>();
   for (int i = 0; i < people.length; i++)
       (people[i].age < 18 ? minorsList : adultsList)
             .add(people[i]);
   minors = minorsList.toArray(people);
   adults = adultsList.toArray(people);
}
```

```
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```

A function value

An infix method call

Сравнение с Java: Параллельная обработка

?

```
val people: Array[Person]
val (minors, adults) = people.par partition (_.age < 18)
```

Средства разработки

JDK 1.7 или 1.8

уже установлен

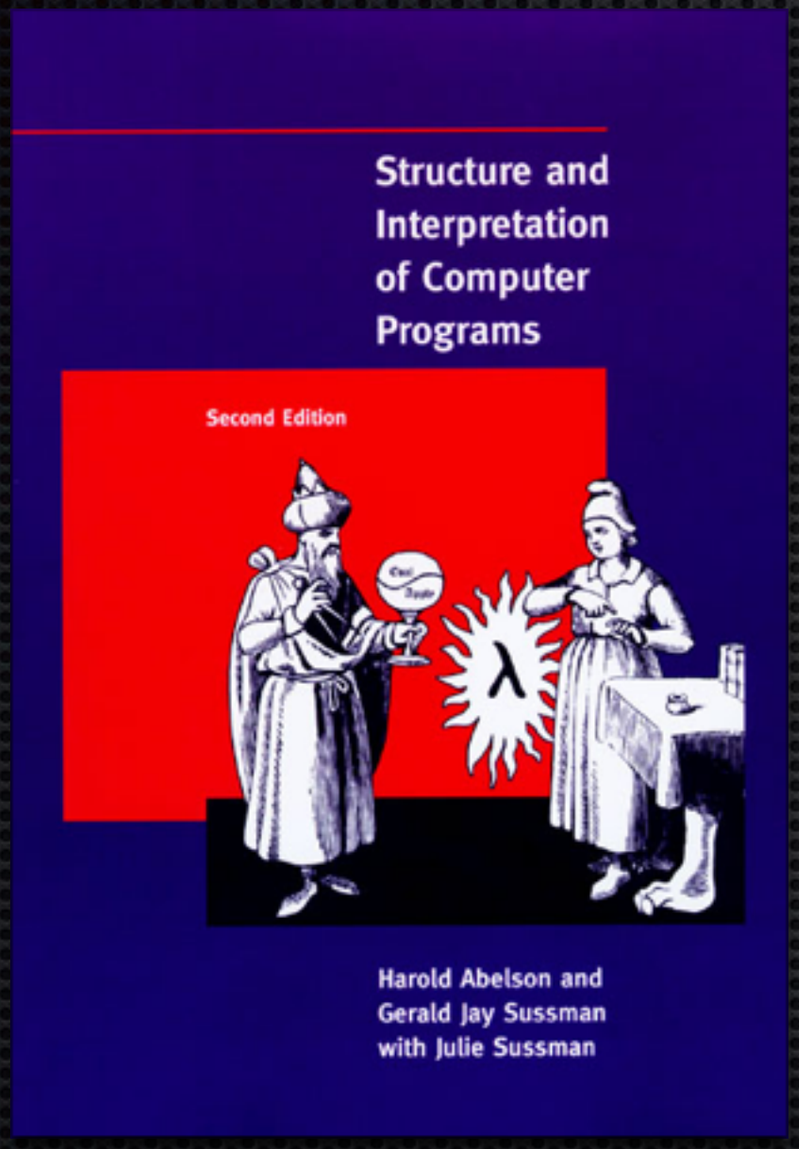
Sbt 0.13.x

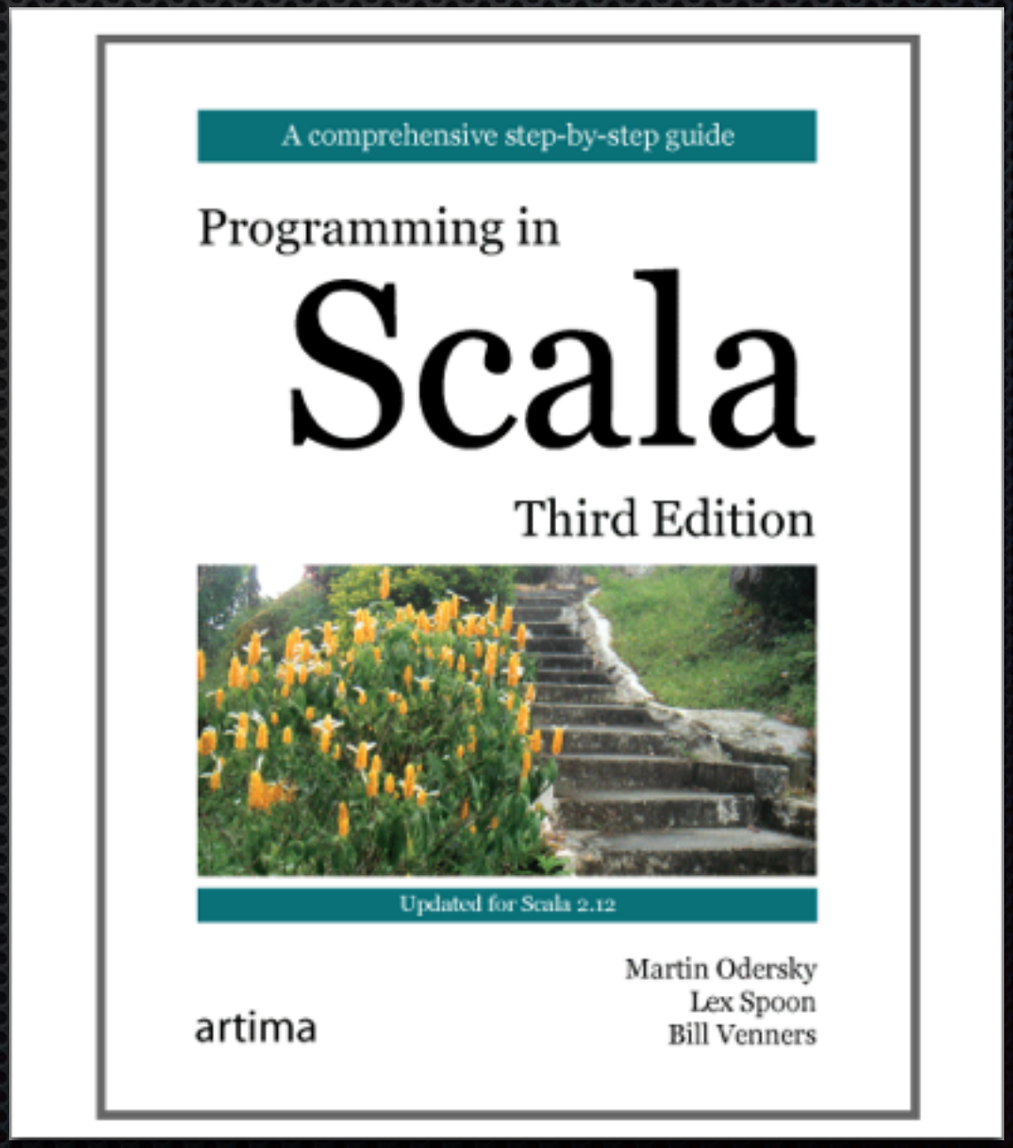
http://www.scala-sbt.org/download.html

Scala IDE for Eclipse (или IntelliJ)

http://scala-ide.org/download/sdk.html

Литература


Structure and
Interpretation
of Computer
Programs
Second Edition
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman


A comprehensive step-by-step guide
Programming in
Scala
Third Edition
Updated for Scala 2.12
artima
Martin Odersky
Lex Spoon
Bill Venners

Краткий обзор языка

Scala - язык, ориентированный на выражения (expression-oriented)

REPL (Read-Eval-Print-Loop)

Основные элементы (объявления):
- выражения: 2 + 2
- константы: val four = 2 + 2
- переменные: var greeting = “Hello, Scala!”
- функции: def incr(x: Int) = x + 1

Классы

Классы похожи на классы Java.
Для инстанцирования используется new.
Существуют особенные классы - case.
У классов Scala не может быть статических членов.

```
class Calculator {
  val brand: String = “Электроника”
  def add(m: Int, n: Int) = m + n
}

val calc = new Calculator
```

Трейты и объекты

Трейты (trait)
Похожи на интерфейсы Java.
Могут содержать конкретные члены.
Для использования - ключевое слово extends (или with - если больше одного)

Объекты (object)
Всё в Scala - объект.
Объекты используются для хранения одного экземпляра класса (Singleton).
Нельзя создать при помощи new.

Пакеты

Пакеты (package) используются для организации кода.

Объявляются вверху файла (как и в Java).

```
package foo.bar
```

Импортируются пакеты:

```
import foo.bar._
```

не обязательно целиком:

```
import foo.bar.MyClass
```

Выполнение вычислений

Составное выражение вычисляется следующим образом:

1. Получить самый левый оператор

2. Вычислить его операнд

3. Применить оператор к операндам

Имя выражения заменяется правой частью его объявления.

Процесс заканчивается, когда результатом становится значение (например, число)

Подстановочная модель применения функции

Объявления могут иметь параметры:

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

Применение функции с параметрами:

1. Вычислить все аргументы (слева направо)

2. Заменить вызовы функции её правой частью, заменяя параметры аргументами

Вычисление сокращает выражение до его значения (если у выражения нет побочных эффектов)

Подстановочная модель лежит в основе λ-исчисления.

Функциональный и нефункциональный подходы (пример)

```
val x = “Hello, World!”
val r1 = x.reverse
val r2 = x.reverse

val r1 = “Hello, World!”.reverse
val r2 = “Hello, World!”.reverse
```

```
val x = new StringBuilder “Hello”
val y = x.append(“, World!”)
val r1 = y.toString
val r2 = y.toString

val r1 = x.append(“, World!”).toString
val r2 = x.append(“, World!”).toString
```

Ссылочная прозрачность

- Выражение **e** ссылочно-прозрачно (referentially transparent) для всех программ **p**, если все вхождения **e** в **p** могут быть заменены на результат вычисления **e** не изменяя поведения **p**.

- Функция **f** - чистая (pure), если выражение **f(x)** ссылочно-прозрачно для всех ссылочно-прозрачных **x**.

```
this.remove(this.findMin).ascending(t + this.findMin)
```

Вызов по имени vs. вызов по значению

Всегда ли можно вычислить выражение?

```
def loop: Int = loop
```

Изменение стратегии вычисления - например, применить функцию к не сокращённым аргументам

Первый способ: вызов по значению (call-by-value)
Второй способ: вызов по имени (call-by-name)

Обе стратегии сокращают выражение до одного и того же значения:
- если выражение состоит из “чистых функций”
- оба вычисления конечны

Вызов по значению вычисляет каждый аргумент только один раз
Вызов по имени не вычисляет аргументы функции, если они не используются

Стратегия вычислений в Scala

- Если CBV выражения e вычисляется, то вычисляется и CBN
- Но не наоборот!

В Scala могут использоваться обе стратегии:

```
def constOne(x: Int, y: => Int) = 1
```

Кроме того: def - CBN; val - CBV

Условные выражения

if-else выглядит так же как в Java, но применяется к выражениям (вычисляется!)

```
def abs(x: Int) = if (x >= 0) x else -x
```

Логические выражения

Состоят из:

- констант: true false
- отрицаний: !b
- конъюнкций: b && b
- дизъюнкций: b || b
- операций сравнения:
  e <= e, e >= e
  e < e, e > e
  e == e, e != e

Блоки и области видимости

Блок в Scala: { ... }

Содержит последовательность объявлений и выражений

Последний элемент в блоке - определяет его значение

Блоки сами по себе являются выражениями

Объявления внутри блока видны только в блоке

Объявления во внешних блоках видны, если не перекрыты

Пример

Каково значение result?

```
val x = 0
def f(y: Int) = y + 1
val result = {
                val x = f(3)
                x*x }+x
```

Code Convention: Запрещается!!!!!!!!

- isInstanceOf и asInstanceOf
- var
- return
- ;
- if (cond) true else false
- print

Code Convention: Приветствуется

- форматирование
- короткие строки
- осмысленные названия
- локальные значения (осмысленные)
- подвыражения

```
this.remove(this.findMin).ascending(t + this.findMin)
```

Про тестирование