Scala for TAPL’ers
Part I

Scala Intro
Objects & Functions
Pattern Matching
For-comprehensions
Implicit conversions

Ilya Sergey
ilya.sergey@cs.kuleuven.be
9 November 2010
Scala

What Java should have been

• Fusion of object-oriented and functional programming
• Designed by Martin Odersky et al, EPFL
• Inspired by Java, Haskell, Erlang, Standard ML

http://scala-lang.org
The goal

• To show what Scala can do
• To give the minimal Scala background for STLC interpreter implementation
• To discuss good Scala programming style
• To show what and where to search for reference
Not a goal

• To give a detailed survey of syntax and SDK
• To tell about all tricks and available tools
• To convince everybody to program in Scala only
Recommended reading

• Scala Language Specification (SLS)
• Scala by example
• Programming in Scala
  • http://www.artima.com/shop/programming_in_scala
• Scala wiki (heaps of examples)
  • http://scala.sygneca.com/
Say hello in Java...

class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
... and in Scala

```scala
object Test {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```
or just

```scala
object Test extends Application {
  println("Hello, world!")
}
```
“Hello World” revised

object Test {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}

- Parameter and variable types go after a name
- Type generics in square braces
- **def** keyword defines a function
- **object** keyword defines a singleton
object Test extends Application {
  println("Hello, world!")
}

trait Application {
  def main(args: Array[String]) = {}
}

Application’s main() method is mixed into the Test object
Scala’s program decomposition

- *Singleton objects* for “static” members
- *Traits* as containers for specific behaviour and interfaces
- *Classes* to combine traits together into particular instances
Scala syntax cheat sheet

• Definitions start with a keyword
  • `class / trait / object name[params](args) extends T { members }`
  • `val / var name : type = ...`
  • `def name[params](args) : type = ...`

• Arbitrary nesting allowed

• Modifiers
  • `abstract` (only with class)
  • `override` (required when overriding)

• Types come after names (and can often be omitted)
Scala REPL

Welcome to Scala version 2.8.0
(Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_20).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("finally")
finally
Demo: Scala REPL

- Use <Tab> for method completion
- Type :help for reference
- Use :load to load a content from file
Functional programming

- Functions are first-class citizens
- Primitive functions:
  - Lambda abstraction
  - Application

```scala
scala> val inc = (x: Int) => x + 1
inc: (Int) => Int = <function>
scala> inc(41)
res1: Int = 42
```
Higher-order functions

\[
\text{List}(1,2,3).\text{map} \{(x:\text{Int}) \Rightarrow x \times 2\}
\]

\[
\text{List}(1,2,3).\text{map} \{x \Rightarrow x \times 2\}
\]

\[
\text{List}(1,2,3).\text{map} \_ \times 2)
\]

Haskell’s section
FP meets OO

• Function = instance of FunctionN trait
• Application = method call

```
trait Function1[-T1, +R] extends AnyRef { self =>
  def apply(v1:T1): R
  override def toString() = "<function>"
}
```

```
scala> object inc extends Function1[Int, Int] {
   |    def apply(x: Int) = x + 1 }
defined module inc

scala> inc(41)  // really, inc.apply(42)
res1: Int = 42
```
Type variance

- Functions are
  - **Covariant** by return type
  - **Contravariant** by parameter type

```scala
trait Function1[-T1, +R] extends AnyRef {
  self =>
  def apply(v1:T1): R
  override def toString() = "<function>"
}
```
Applying objects

• Every instance, which have `apply()` method may be *applied* as a function

```scala
val list = List(1,2,3)
```

• Do not confuse with instance creation!

```scala
val list = new List {...}
```
Methods in Scala

• Methods may be called in infix notation
• The precedence is similar to arithmetic expressions
• There is no operator overloading: every “operator” is a method

scala> List(1,2).++(List(3,4))
res9: List[Int] = List(1, 2, 3, 4)

scala> List(1,2) ++ List(3,4)
res10: List[Int] = List(1, 2, 3, 4)
By this time you know

• How to create functions in Scala
• How to create lists

And also you know Haskell

Let’s implement something!
1. Given a list of integers. Implement a function that removes all odd numbers from it

2. Given a list of integers. Compute a list of appropriate factorials.

**Hints**

You can define auxiliary functions:

Recursive functions need return type!

```scala
scala> def fact(n: Int): Int = if (n == 0) /* ... todo */

Use higher-order functions:  map, filter
Side-note: *partially-defined functions*

**Haskell**

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (\text{head} : \text{tail}) &= 1 + \text{length } \text{tail}
\end{align*}
\]

**Scala**

```scala
def length : List[_] => Int = {
  case Nil => 0
  case head :: tail => 1 + length(tail)
}
```

• For recursive definitions type is mandatory
• Nil is an object for an empty list
• h :: t is a decomposition of a list to a head and tail
For-expressions (I)

• Iterating through iterable objects
• Producing new collections

```scala
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere)
    println(file)
```
For-expressions (II)

• Iterations may be nested

• Filters

```scala
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines

def grep(pattern: String) =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
  } println(file +": "+ line.trim)

grep(".*gcd.*")
```
For-expressions (III)

- Producing a new collection

```scala
code snippet
```
Exercise

- Implement quicksort for integer lists in Scala using for-expressions

```scala
def qsort: List[Int] => List[Int] = {
  case Nil => /* todo!*/
  case h :: t => /* todo! */
}
```

Hints

- For-comprehensions are expressions!
- Use `++` to concatenate lists
- Use `yield` statement to produce a new collection
Implicit conversions

• Decorator pattern

```scala
implicit def stringWrapper(s: String) = 
  new RandomAccessSeq[Char] {
    def length = s.length 
    def apply(i: Int) = s.charAt(i)
  }
```

```
scala> stringWrapper("abc123") exists (_.isDigit)
res0: Boolean = true

scala> "abc123" exists (_.isDigit)
res1: Boolean = true
```
STLC modeling

• Let’s model something

  \[ t ::= x \quad \text{variable} \]
  \[ \lambda x. \ t \quad \text{abstraction} \]
  \[ t \ t \quad \text{application} \]
In Java

package syntax;

interface Term {}

class Var implements Term {
    public Var(String n){_name = n;}
    private String _name;
    public void setName(String n) {
        _name = n;
    }
    public String getName() {
        return _name;
    }
}

// Don't have time to write down the rest...
In Scala

```scala
trait Syntax {
  trait Term {
    case class Var(name: Name) extends Term
    case class Abs(x: Name, body: Term) extends Term
    case class App(fun: Term, arg: Term) extends Term
  }
}
```

[for now, assume Name = String]

- Consider case class `Var(name: Name)` extends `Term`
  - defines class `Var`, subclass of `Term`
  - has one member, `val name: Name`
  - construction = like calling `def Var(name: Name): Var`
  - “deconstruction” = pattern matching
Case Classes

- Convenient way of defining a class with
  - its public fields, and
  - default constructor
- Instantiation may omit the `new` keyword
- Encapsulation = ok
  - fields can be overridden
- Pattern matching
Exercise: mental parsing

- Encode the following Lambda terms:
  - \( \lambda x. x \)
  - \( \lambda y. \lambda x. y \)
  - \( (\lambda x. x x) (\lambda x. x x) \)

```scala
case class Var(name: Name) extends Term
case class Abs(x: Name, body: Term) extends Term
case class App(fun: Term, arg: Term) extends Term
```

[for now, assume Name = String]
Pattern matching: example

- A structured way to take data apart
  - Not everything can be modeled using OO’s late binding
  - A point doesn’t “know” how to move a robot...

- Match an expression \( p \) with patterns in cases, top to bottom (‘if ...’ = guard)

```scala
case class Point2D(x: Double, y: Double)

def moveHeavyRobot(p: Point2D) = p match {
  case Point2D(0, 0) => // optimise: do nothing
  case Point2D(x, y) if x == y => moveDiag(x)
  case Point2D(x, y) => doMove(x, y)
}
```
Pattern Matching

See 7.1 and 7.2 of Scala by Example

- pattern:
  - case class constructor w/ patterns as args
  - pattern variable (lower case!)
    - may only occur once in pattern!
  - don’t care (_)
  - literals (1, “abc”)
  - constant identifiers (upper case)
- guard: arbitrary expression
Example: *Option class*

```scala
def show(x: Option[String]) = x match {
  case Some(s) => s
  case None => "?"
}
```

- Indicates an optional value
- Type check instead of silent null passing
Exercise
(from Scala by Example)

Consider the following definitions representing trees of integers:

Exercise 7.2.2 Consider the following definitions representing trees of integers:

```scala
trait IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, smaller: IntTree, greater: IntTree) extends IntTree
def contains(t: IntTree, v: Int): Boolean = t match {
  case EmptyTree => false
  case Node(x, _, _) if v == x => true
  case Node(x, l, _) if v < x => contains(l, v)
  case Node(x, _, r) => contains(r, v)
}
```
Next time

- Beta-reduction vs Closures
- Parser combinator basics
- Monadic computations
Homework

• [FP] Add checkdiff method to the List class to check that all elements of a list are different

• [case classes] Implement a data structure for electrical circuits (for resistors in series or in parallel) to compute their resistance by Ohm’s law
IntelliJ IDEA Demo

• How to install the plugin
• How to set up Scala libraries
• How to compile, run and debug a program