Functional Programming and the Scala Language

Lecture 5

Eugene Zouev
Innopolis University
Spring Semester 2018

Pattern Matching

Basic Scala Collections

Match Expression

No `switch` statement - `match` expression instead

Example from the previous lecture

```
val firstArg = if (args.length>0) args(0) else ""

val friend = firstArg match {
  case "salt"  => "pepper"
  case "chips" => "salsa"
  case "eggs"  => "bacon"
  case _       => "huh?"
}
println(friend)
```

The main point:
What can be specified as "case"?

Match Expression: cases

1. Constants (literals)

```
val ch: Char
  ...
val sign = ch match {
  case '+' => 1
  case '-' => -1
  case _   => 0
}
```

2. Variables

```
var str: String
  ...
val sym = str(i) match {
  case '+' => 1
  case '-' => -1
  case ch  => Character.digit(ch,10)
}
```

Here, ch matches any character except '+' & '-'

Match Expression: cases

3. Types (!)

```
val obj = ...
  ...
val res = obj match {
  case Int    => "obj is Integer"
  case String => "obj is String"
  case BigInt => "obj is BigInt"
  case _      => "Some other type"
}
```

Here, match is actually a generalization of Java's instanceOf

4. Type & Variables

```
val obj = ...
  ...
val res = obj match {
  case x: Int    => x
  case s: String => Integer.parseInt(s)
  case _: BigInt => Int.MaxValue
  case _         => 0
}
```

If obj is of type Int, then x matches obj as of type Int.
Else if obj is of type String, then s matches obj as of type String.
Etc…

Match Expression: cases

5. Tuples

```
val roots = QRoots(a,b,c)
  ...
val res = roots match {
  case (_, 0.0)        => "The 2nd root is zero"
  case (x, y) if x==y  => "The single root"
  case (_, _)          => "Two different roots"
}
```

First case: we don't care about 1st tuple element; we check if its 2nd element matches 0.0 literal.

Second case: x & y match corresponding tuple elements. Here we need them to compare for equality (in guard clause).

6. Arrays (and other collections)

```
anArray match {
  case Array(0)              => "array with 0"
  case Array(x, y) if x==y   => "array with two equal elements"
  case Array(_, _)           => "array with two elements"
}
```

Constructor patterns

Match Expression: cases

What's wrong with the code?

```
anArray match {
  case Array(_, _)          => "array with two elements“
  case Array(x, y) if x==y => "array with two equal elements"
}
```

The second case is never considered - because the first one "covers" the second!

Or: cases are processed in order of their appearance in the code.

The consequence:
Put more specialized cases first

You might know the similar situation with exception handlers in C++

Match Expression: cases

Cases as function literals (!)

We will come back to this feature later

```
val obj = ...
  ...
val res = obj match {
  case x: Int     => x
  case s: String => Integer.parseInt(s)
  case _: BigInt => Int.MaxValue
  case _          => 0
}
```

Cases in match can be considered as multiple "entries" to functions where constructs after each case are treated as function parameters, and expressions after => as function bodies...

match: Pattern Matching in Full

Simplified expression grammar

```
Expression : Variable
           | Number
           | UnaryOperator Expression
           | Expression BinaryOperator Expression
```

How the grammar could be represented programmatically:

```
abstract class Expr
case class Var(name: String) extend Expr
case class Number(num: Double) extend Expr
case class UnOp(op: String, arg: Expr) extend Expr
case class BinOp(op: String, left: Expr, right: Expr) extend Expr
```

Expression example

```
BinOp("*",
      Var("a"),
      BinOp("+",
            Var("b"),
            Number(2)))
```

a*(b+2)

match: Pattern Matching in Full

The task: to implement **expression simplification** using a few obvious rules

Transformation rules ("algebra"):

-(-*e*) ➔ *e* //double negation
e + 0 ➔ *e* // adding zero
e * 1 ➔ *e* // multiplication by one

where *e* is an expression

```
UnOp("-", UnOp("-", e))      => e
BinOp("+", e, Number("0")) => e
BinOp("*", e, Number("1")) => e
```

```
def simplify(expr: Expr): Expr =
  expr match {
    case UnOp("-", UnOp("-",expr))   => expr
    case BinOp("+",expr,Number("0")) => expr
    case BinOp("*",expr,Number("1")) => expr
    case _                           => expr
}
```

match: Full Coverage by Cases

The problem with case class hierarchies:

```
abstract class Expr
case class Var(name: String) extend Expr
case class Number(num: Double) extend Expr
case class UnOp(op: String, arg: Expr) extend Expr
case class BinOp(op: String, left: Expr, right: Expr) extend Expr
```

```
def simplify(expr: Expr): Expr =
  expr match {
    case UnOp("-", UnOp("-",expr))    => expr
    case BinOp("+",expr,Number("0")) => expr
    case BinOp("*",expr,Number("1")) => expr
    case _                            => expr
}
```

Common behavior

Suppose a user has added a new class to the hierarchy in his/her own program:

```
case class ArrayElem(arr: Expr, index: Expr) extend Expr
```

In that case, simplify cannot handle all possible cases.

match: Full Coverage by Cases

The problem with case class hierarchies:

```
abstract class Expr
case class Var(name: String) extend Expr
case class Number(num: Double) extend Expr
case class UnOp(op: String, arg: Expr) extend Expr
case class BinOp(op: String, left: Expr, right: Expr) extend Expr
```

```
case class ArrayElem(arr: Expr, index: Expr) extend Expr
```

```
def simplify(expr: Expr): Expr =
  expr match {
    case UnOp("-", UnOp("-",expr))   => expr
    case BinOp("+",expr,Number("0")) => expr
    case BinOp("*",expr,Number("1")) => expr
    case _                           => expr
}
```

Common behavior will cover the case with `ArrayElem`, but it's impossible to specify non-default processing for it

The solution: explicitly prohibit adding new classes to the hierarchy:

```
sealed abstract class Expr
```

match: Full Coverage by Cases

The problem with case class hierarchies:

```
sealed abstract class Expr
case class Var(name: String) extend Expr
case class Number(num: Double) extend Expr
case class UnOp(op: String, arg: Expr) extend Expr
case class BinOp(op: String, left: Expr, right: Expr) extend Expr
```

```
def describe(e: Expr): String =
  e match {
    case Number(_) => “a number”
    case Var(_)    => “a variable”
  }
```

```
Warning: match is not exhaustive!
missing combination        UnOp
missing combination        BinOp
```

In case of sealed, compiler is able to detect missing cases

Pattern Matching: Assignment 1

Write the class hierarchy for expressions presented on the last lecture. Test the function `simplify` against your own examples.

Write the function `simplifyAll` that should perform the same actions for the whole expression but not for its top-level parts.

Hint: the function should recursively call itself.

In addition to simplifications defined previously, implement the following transformations:

```
+e    → e     // identity operation
e * 0 → 0     // multiplication by zero
e + e → e*2   // replace addition for mult.
```

Test `simplifyAll` against a reasonable set of expressions.

Pattern Matching: Assignment 2

Suppose there is an expression written in accordance with the grammar from the lecture with two exceptions: **only binary operators** and operands are **only integer constants** (no variables).

The simplified grammar:

```
abstract class Expr
case class Number(num: Integer) extend Expr
case class BinOp(op: String, left: Expr, right: Expr) extend Expr
```

Example of expression:

```
1+(2-3*4)/5
```

Operator signs:

```
+   -   *   /
```

Remark 1: operator / means integer division

Remark 2: +, - have lower precedence;
*, / have higher precedence

Pattern Matching: Assignment 2

The assignment is to write a Scala program reading the input string with an expression written in accordance with the grammar and **calculate the value of the expression**.

Example: the input expression like

`1+(2-3*4)/5` should be calculated to `-1`

The solution should consist of (at least) two functions:

- The first function (`read`) should read the expression from the input string and build the corresponding tree with nodes of types `BinOp` and `Number`.
  Perhaps you will need some nested functions within `read`.
- The second function, `calculate`, should perform actual calculations. The function **should use pattern matching mechanism**.

Pattern Matching: Assignment 2

`read`: a draft

Organize a loop over characters taken from the input string. On each iteration check the character:

- If it is '(' then skip it and invoke `read()` function recursively. It should return an `Expr`. After returning from the function, check for final ')'.
- If it is a **digit** then scan the source until the number ends, create and return `Number(`*`substring`*`)`
- Otherwise, invoke `read()` function recursively, get *`rightOp`* of type `Expr` as the result. Then check for an operator sign, and if there is one, keep it and invoke `read()` again for the right operand. After that create and return `BinOp(`*`sign`*`,` *`leftOp`*`,` *`rightOp`*`)`.
- Remarks: take care about **operator precedence**, (perhaps you'll need two nested functions for additive and multiplicative operators) and about **end of source string**. Assume that the input string always contains **syntactically correct expression**.

Pattern Matching: Assignment 2

calculate: a draft

```
def calculate(expr: Expr): Integer =
  expr match {
    case Number(str) => Integer.parseInt(str)
    case BinOp("+",expr1,expr2) =>
           calculate(expr1)+calculate(expr2)
    case BinOp("*",expr1,expr2) =>
           calculate(expr1)*calculate(expr2)
    ...
    case _ => ???
}
```

BTW, this is not an informal remark:
this is **the name of a real method**
throwing the exception like
`NotImplemented` ☺

Basic Scala Collections:

`Array`

`List`

Scala Collections: `Array` & `List`

Some common info about arrays & lists in Scala

- Both are fundamental and popular data structures in Scala; they don't differ much from similar data structures in other languages (e.g., C++, C# etc.).
- Both contain elements of some type. The type is **the same** for all elements: these structures are **homogeneous**.
- Both are **generic** data structures: i.e., they are parametrized by a type of their elements.
- Both are implemented **as classes**.
- Both are the part of the standard Scala library.
- Both have a big set of operations (implemented as class methods and/or operators)

Scala Collections: Array & List

What's the **common differences** between arrays & lists?

- Advantages & disadvantages; where and when to use array/list instead of list/array?
- How arrays and lists are represented internally?

You should know that – or read any textbook about data structures ☺

Some differences between arrays & lists **in Scala**

- Arrays are **mutable** in Scala; lists are **immutable**.
- The implementation of arrays is based on the Java's array implementation. This gives full interoperability between Java & Scala programs (i.e., a Scala program can use arrays declared in a Java program, and **vice versa**).
- Lists are implemented "from scratch" in Scala

Scala Collections: Array

How to declare and create an array?

Conventional (Java-like) way

```
val greets = new Array[String](3)
```

How to assign to array elements?

```
greets(0) = "Hello"
greets(1) = ", "
greets(2) = "world!"
```

Conventional (Java-like) way

Internally gets converted to:

```
greets.update(0, "Hello")
greets.update(1, ", ")
greets.update(2, "world!")
```

Why we use parentheses instead of square brackets?
Quick explanation:
- Arrays in Scala are just **class instances**. Therefore, any access to a class instance is actually **a call to a method**. In case of arrays there are two methods: `apply` for getting array elements, and `update` for modifying them.

```
val s = greets(0)
```

```
val s = greets.apply(0)
```

Scala Collections: Array

How to declare and create an array?

Scala way

```
val nums = Array("zero","one","two")
```

No need to mention String as the type of elements, and the array size:

- Compiler understands (infers, deduces) the type of array elements from (types of) initializers
- Compiler deduces the size of the array from amount of initializers from the list

Why don't we write new keyword?

- Here, compiler doesn't use Array's constructor for creating the new instance; instead, it uses apply factory method for creation.

The full form:

```
val greets = Array.apply("zero","one","two")
```

Scala Collections: Array

What is "factory method"?

Class implementing the array

```
class Array[T] ...
{
  ...
  apply
  update
  ...
}

object Array
{
  apply
}
```

Methods implementing array functionality

There are very many methods that perform various useful operations on arrays; see Scala documentation.

Array companion object

Factory method for creating array instances

(Similar to "static" methods in C++/Java/C#)

Scala Collections: Array

Arrays are **mutable objects**

```
val greets = Array("Hello", ", ", "world!")
```

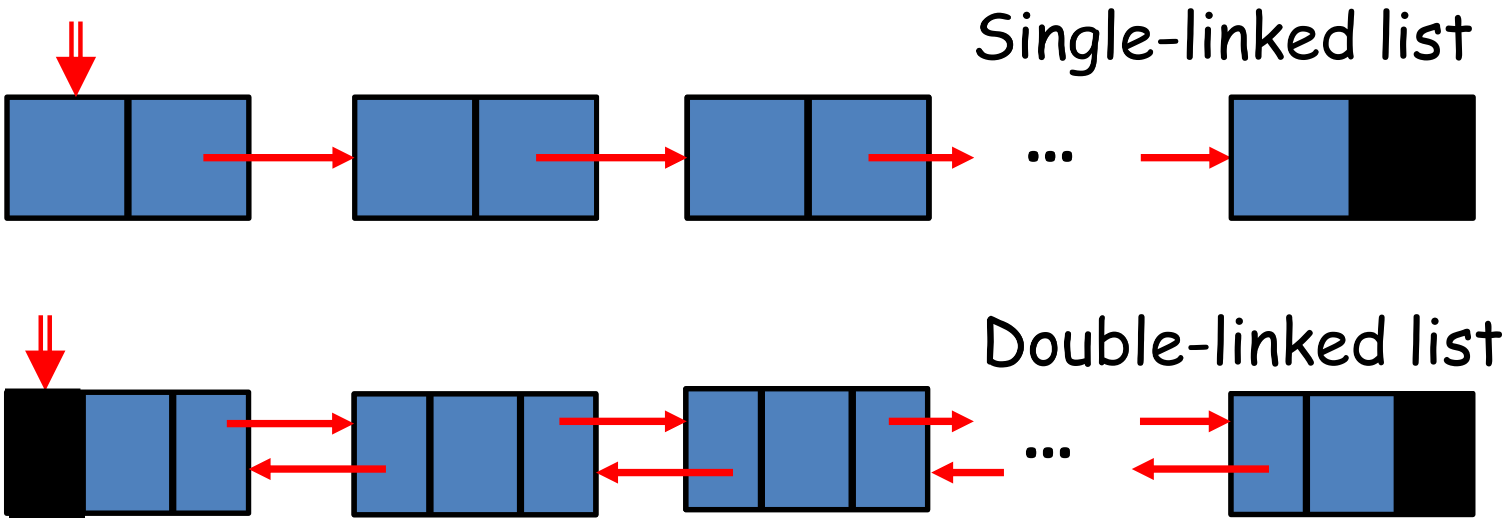
After array was created, it’s impossible to change its type and size. However…

```
greets(2) = "Bob!"
```

…it’s quite possible to change values of its elements.

Scala Collections: `List`

- Do you know what's the difference between arrays and lists in general?
- Do you know how lists are represented internally?



Later we will see how lists are represented in Scala
(they have *recursive* structure)

Scala Collections: List

How to declare and create a list?

```scala
val fruit = List("apples", "oranges", "pears")
```

```scala
val fruit: List[String] =
 List("apples", "oranges", "pears")
```

```scala
val nums = List(1, 2, 3, 4)
```

```scala
val nums: List[Int] =
          List(1, 2, 3, 4)
```

```scala
val matrix = List( List(1, 0, 0),
                   List(0, 1, 0),
                   List(0, 0, 1) )
```

```scala
val matrix: List[List[Int]]
          = List( List(1, 0, 0),
                  List(0, 1, 0),
                  List(0, 0, 1) )
```

```scala
val empty = List()
```

```scala
val empty: List[Nothing] = List()
```

Scala Collections: `List`

- Lists are **generic** structures
- Lists are **homogeneous** structures
- Lists are **immutable** structures
- Lists are **covariant** structures

Will consider variance later

```
val empty = List()
```

Empty lists are represented as `List()` (or as `Nil`, but **not** as `()`) and have type `List[Nothing]`

`Nothing` is the lowest type in the Scala type hierarchy (will see & discuss later)

Scala Collections: `List`

Basic operators on lists

`::` infix binary operator for ***cons***tructing lists

(the similar operator is called `cons` in other functional languages; came from **Lisp** initially)

```
element :: list
```

Common form: this is a list where *`element`* is its first element, and elements from *`list`* go after it

Examples

```
val example0 = Nil
```

```
val example1 = List(1)
```

Gets converted to:

```
val example1 = 1 :: Nil
```

```
val example3 = List(1,2,3)
```

Gets converted to:

```
val example3 = 1 :: (2 :: (3:: Nil))
```

Scala Collections: `List`

Basic operators on lists

The `::` operator is **right associative**. This means that the construct like

```
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
```

is equivalent to

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

…and in turn is the same as

```
val nums = List(1, 2, 3, 4)
```

The common rule taken in Scala: if the name of an operator ends with “`:`”, then the operator has **right associativity**. ☺☺

Scala Collections: List

Basic operators on lists

head the method returns the first element of the list

```
val nums = List(1, 2, 3, 4)
val f = nums.head  // returns 1
```

tail the method returns the list starting from the second element of the initial list

```
val nums = List(1, 2, 3, 4)
val t = nums.tail  // returns (2, 3, 4)
```

isEmpty the method returns **true** if the list is empty, and **false** otherwise

```
val nums = List(1, 2, 3, 4)
val e = nums.isEmpty  // returns false
```

Scala Collections: List

Example & assignment

Sorting list elements by insertions

The idea is as follows: in order to sort a non-empty list represented as `x :: xs` its tail `xs` gets sorted first, and then the first element `x` is inserted to the appropriate position of the result.

```
def isort(xs: List[Int]): List[Int] =
  if (x.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]) : List[Int] =
  if (xs.isEmpty || x <= xs.head) x::xs
  else xs.head :: insert(x, xs.tail)
```

The assignment is to **test** the algorithm on some real list consisting of random-generated integer values and **estimate the complexity** of the algorithm.