INFO0054-1
Programmation Fonctionnelle

Chapter 02: Higher-order programming and functional data structures

Christophe Debruyne

(c.debruyne@uliege.be)

References

- Chapter 2: Getting started with functional programming in Scala.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

- Chapter 3: Functional data structures.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

Overview

- Higher-order programming
  - Function literals
  - Monomorphic and polymorphic functions
  - Functions as input, functions as output
- Functional data structures
  - Algebraic data types
  - ADTs: Singly linked lists, binary trees, and tuples
  - Variances
  - Pattern matching
  - Data sharing

Part 01
Higher-order programming

Higher-order functions

<u>Functions are values!</u>

- They can be assigned to variables
- They can be stored in data structures
- They can be passed as arguments
- They can be returned as a result
- …

For example:

```
charges.reduce((p1, p2) => p1.combine(p2))
```

Illustrating higher-order programming by optimizing our module

Our function fac is recursive, but it is not tail recursive. A call is said to be in tail position if the caller does nothing other than return the value of the recursive call.

In FP languages, tail recursion is optimized by compiling the code as an iteration using loops.

It is thus a good practice to rewrite programs so that they become tail recursive *when possible*.

```
object FactorialModule:
  def fac(n: Int): Int =
    if(n == 0) 1
    else n * fac(n - 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Not tail recursive as the result of the recursive call is still used in a multiplication.

Illustrating higher-order programming by optimizing our module

Our function `fac` now has an inner function `iter` (or local definition).

`iter` is tail recursive and `fac` relies on that inner function to compute the factorial.

Notice that the inner function uses an additional variable that stores intermediate results, which we call an accumulator.

```
object FactorialModule:
  def fac(n: Int): Int =

    def iter(n: Int, acc: Int): Int =
      if(n == 0) acc
      else iter(n - 1, n * acc)

    iter(n, 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

Illustrating higher-order programming by optimizing our module

FP programming languages are good at detecting tail recursion and subsequently optimize those, but it is up to you, the developer to inform the compiler that a function is tail recursive.

If you want to test whether your function is tail recursive, you can add the `@annotation.tailrec` annotation. Scala will throw an error if your function is not tail recursive.

```scala
object FactorialModule:
  def fac(n: Int): Int =
    @annotation.tailrec
    def iter(n: Int, acc: Int): Int =
      if(n == 0) acc
      else iter(n - 1, n * acc)

    iter(n, 1)

  private def formatFac(n: Int) =
    val message = "The factorial of %d is %d"
    message.format(n, fac(n))

  def main(args: Array[String]): Unit =
    println(formatFac(5))
```

But...

- If tail recursion is optimized by transforming it into an iteration with loops, that means the resulting code has side effects! Doesn't that go against the idea of functional programming?
- One can argue that, yes. But there are always side effects:
  - Manipulation of the various stacks
  - Refreshing the screen
  - The garbage collector
  - ...
- We will analyze the pureness of our functions using RT and "ignore" all things that happened behind the scenes such as code optimization and garbage collection.

Higher-order programming: a 2nd example

helper takes as input a binary function.

```scala
object MathModule:

  private def helper(n: Int, acc: Int, op: (Int, Int) => Int): Int =
    @annotation.tailrec
    def h(n: Int, acc: Int): Int =
      if(n == 0) acc
      else h(n - 1, op(n, acc))
    h(n, acc)

  def fac(n: Int): Int = helper(n, 1, _ * _)

  def trianglenumber(n: Int): Int = helper(n, 0, _ + _)

  def main(args: Array[String]): Unit =
    val l = List(0, 1, 2, 3, 4, 5)
    println(l.map(fac))
    println(l.map(trianglenumber))
```

Both fac and trianglenumber are written in terms of helper and provide their respective functions as anonymous functions. Anonymous functions do not have a name.

We map over the list and apply our functions to each element.

Output:

```
$ scala MathModule.scala
List(1, 1, 2, 6, 24, 120)
List(0, 1, 3, 6, 10, 15)
```

Higher-order programming: a 2nd example

The expression `_ * _` is syntactic sugar for `(a, b) => (a * b)`.

The following are all equivalent:

- `(a, b) => (a * b)`
- `(a, b) => {a * b}`
- `(a, b) => a * b`

But only the curly braces allow you to place multiple statements. Remember, curly braces group statements into a block.

Higher-order programming: a 2nd example

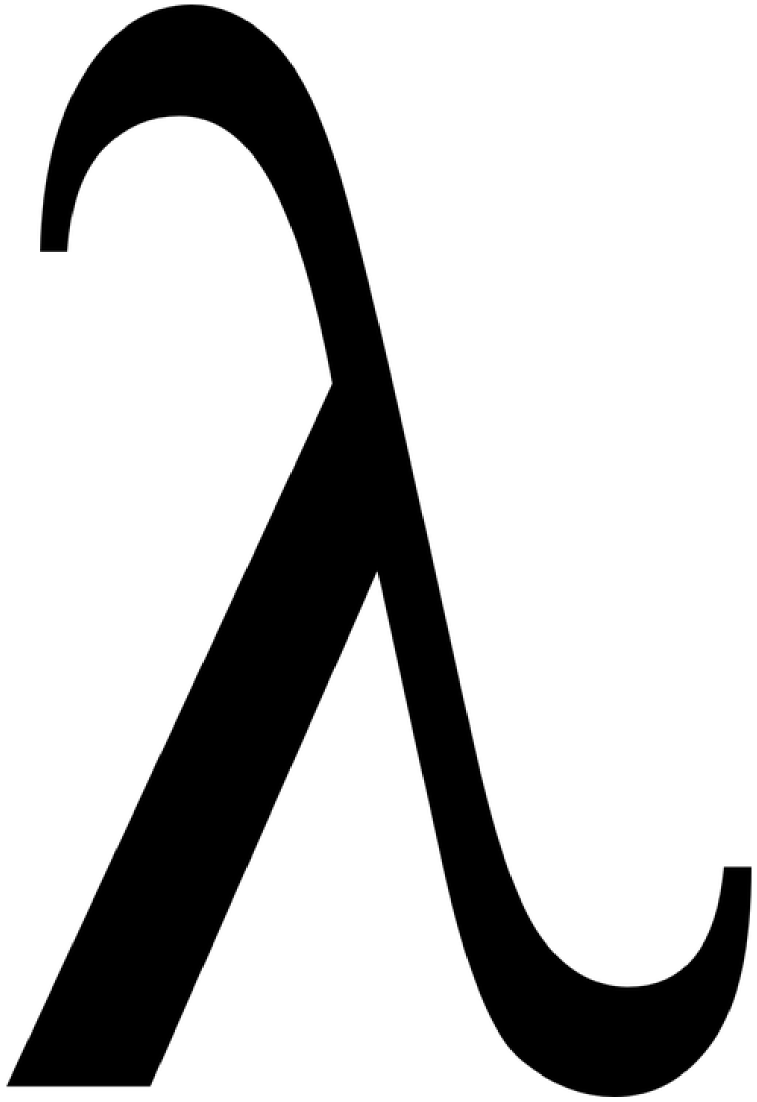
Note that Scala is *quite good* at inferring the input and return types of functions from its context.

An example of an anonymous function, or lambda.

```
scala> List(1, 2, 3, 4).map(x => x * x)
val res0: List[Int] = List(1, 4, 9, 16)

scala> List(1, 2, 3, 4.0).map(x => x * x)
val res1: List[Double] = List(1.0, 4.0, 9.0, 16.0)

scala> x => x * x
-- [E081] Type Error: ----------------------------------------------------------
1 |x => x * x
  |^
  |Missing parameter type
  |
  |I could not infer the type of the parameter x.
1 error found
```

By the way…

- In Computer Science, anonymous functions are also known as function literals, lambda functions, lambda expressions, lambdas, …
- Anonymous functions are functions – or more general, methods – that are not bound to an identifier (i.e., name).
- To read: What's in a Lambda? by Murtaza Ali. PDF also available on eCampus.
    - While written for Python, the article describes the pros and cons of anonymous functions in practice.



Source: OpenIcons on Pixabay

Function literals

```
scala> (x: Int, y: Int) => x ^ y
val res7: (Int, Int) => Int = Lambda$1691/0x000000080115b3c8@92fa950
scala> ((x: Int, y: Int) => x ^ y)(4, 5)
val res8: Int = 1
scala> ((x: Int, y: Int) => x ^ y).apply(4, 5)
val res9: Int = 1
```

- The definition of functional literals in Scala is syntactic sugar of the definition of special function objects with an `apply` method. For instance, `(x: Int, y: Int) => x ^ y` is the same as:

```
new Function2[Int, Int, Int]:
    def apply(x: Int, y: Int): Int = x ^ y
```

- When a function literal is not bound to a variable, we have an anonymous function. We can bind functional literals to variables as follows.

```
val xor = new Function2[Int, Int, Int]:
    def apply(x: Int, y: Int): Int = x ^ y
```

By the way:
def vs val

What's the difference between def and val (when declaring functions)?

- A def evaluates on call (lazy evaluation, see later) and evaluates on every call.
- A val evaluates when defined.

```
def plus1: Int => Int = { println("O") ; _ + 1 }

val plus2: Int => Int = { println("X") ; _ + 2 }

----------

scala> :load Test.scala
X
def plus1: Int => Int
val plus2: Int => Int =
Lambda$1734/0x00000008012ee400@5f0369af

scala> plus1(1)
O
val res0: Int = 2

scala> plus1(2)
O
val res1: Int = 3

scala> plus2(1)
val res2: Int = 3

scala> plus2(2)
val res3: Int = 4
```

plus2 is evaluated when defined.

plus1 is evaluated every time it is called.

Polymorphic functions

- Monomorphic functions operate on only one type of data.
  - E.g., `fac` operates on arguments of the type `Int`.

- Polymorphic functions operate on arguments of different types.
  - Ideally any type, but at least more than one type.
  - Also known as parametric polymorphism.

- To exemplify polymorphic functions, we will transform a monomorphic function into a polymorphic version.
  - We will observe some limitations of Scala's type inference and propose a new solution.

Watch out: this notion of polymorphism is different from polymorphism in Object-Oriented Programming, where it refers to *inheritance*.

Example: a monomorphic function

```
object HOModule:
    def findIndexOfFirst(arr: Array[String], key: String): Int =
        @annotation.tailrec
        def loop(n: Int): Int =
            if(n >= arr.length) -1
            else if (arr(n) == key) n
            else loop(n + 1)
        loop(0)
```

`findIndexOfFirst` returns the first index in an array where the key occurs. If the key is not found, -1 is returned. This version of `findIndexOfFirst` is monomorphic as it is defined to look for a string in an array of strings.

```
scala> HOModule.findIndexOfFirst(Array("a", "b", "c", "b", "a"), "b")
val res0: Int = 1
```

Example: a monomorphic function

Problem? Creating similar functions for different types will result in functions whose implementations are almost identical.

```scala
def findIndexOfFirst(arr: Array[String], key: String): Int =
    @annotation.tailrec
    def loop(n: Int): Int =
        if(n >= arr.length) -1
        else if (arr(n) == key) n
        else loop(n + 1)
    loop(0)

def findIndexOfFirst(arr: Array[Int], key: Int): Int =
    @annotation.tailrec
    def loop(n: Int): Int =
        if(n >= arr.length) -1
        else if (arr(n) == key) n
        else loop(n + 1)
    loop(0)
```

```
scala> HOModule.findIndexOfFirst(Array(1, 2, 1, 2, 3), 3)
val res0: Int = 4

scala> HOModule.findIndexOfFirst(Array("a", "b", "c"), "c")
val res1: Int = 2
```

Example: a polymorphic function

- A polymorphic function uses a comma-separated list of type parameters, which are surrounded by square brackets and placed after the name of the function. You can choose the names of your type parameters. By convention, they are usually kept short.
- Instead of hard-coding an array of strings, we take the type parameter A for both an array of type A and a key of type A.

```scala
def findIndexOfFirst[A](arr: Array[A], key: A): Int =
    @annotation.tailrec
    def loop(n: Int): Int =
        if(n >= arr.length) -1
        else if (arr(n) == key) n
        else loop(n + 1)
    loop(0)
```

- The type parameters introduce type variables that we can use in the type signature, which defines the inputs and output of a function (or method) including the number of arguments, the order of arguments, and the types of the arguments. Type signatures are used to determine which function definition needs to be applied.

By the way...

<u>In Scala,</u>

All objects have a function `equals`, which can be overridden (*). The goal of this function is to test the equality of object w.r.t. values.

We can compare object with `equals`. Know that `a equals b` corresponds with `a.equals(b)`.

We also have access to the `==` operator. This operator depends on the `equals` function of the first operand, <u>but properly deals with null values</u>.

We also have the `eq` operator, which checks the <u>equality of references</u>.

```scala
case class Cat(name: String) { }
val a = Cat("Gaston")
val b = Cat("Gaston")
val c = null
a == b        // true, values are the same
a equals b   // true, values are the same
a eq b        // false, objects are different
c == a        // false, values are different
c equals a   // error, c is null!
c eq a        // false, refs are different
```

(*) Watch out: There are some technical quirks surrounding the `equals` *and hashCode functions of objects (in Java and the JVM). For primitive values have their* `equals` *and hashCode functions defined. The Scala compiler also generates these functions for case classes. For "normal" classes, however, one should override these two functions. We may cover these technicalities in more detail later in this course, but you will cover this in your course on object-oriented programming (OOP). This technical detail may be relevant for your project, but it is not for this course.*

Example: calling HOFs with lambdas

- Let us refine `findIndexOfFirst` so that it takes as input an array of type A and a function applied to objects of type A that represent a condition. This function returns thus values of the type Boolean.

```scala
def findIndexOfFirst[A](arr: Array[A], cond: A => Boolean): Int =
    @annotation.tailrec
    def loop(n: Int): Int =
        if(n >= arr.length) -1
        else if (cond(arr(n))) n
        else loop(n + 1)
    loop(0)
```

```
scala> HOModule.findIndexOfFirst(Array(1, 2, 1, 2, 3), (x) => { x == 3 })
val res1: Int = 4
```

Lambda

Example: returning functions

- We will now exemplify the returning of functions with partial function application. Partial function application allows us to reduce the number of arguments (arity) of a function by binding some arguments and returning a new function with a smaller arity.

- Let us look at the following signature

```
def partial[A, B, C](a: A, f: (A, B) => C) : B => C
```

- The `partial` function has
  - Three type parameters.
  - Takes as input
    - an object `a` of the type `A`
    - a function `f` that takes as input an `A` and a `B` and returns a `C` .
  - And returns a function that takes as input a `B` and returns a `C`.

- How would we implement `partial`?

```
def partial[A, B, C](a: A, f: (A, B) => C) : B => C =
    (b) => f(a, b)
```

A type annotation for b is not necessary. Scala can infer b's type from the context.

Example: returning functions

Let's use `partial` to create a function that returns +1.

```
def partial[A, B, C](a: A, f: (A, B) => C) : B => C =
        (b) => f(a, b)

----------
scala> val plus = (x: Int, y:Int) => x + y
val plus: (Int, Int) => Int = Lambda$1696/0x000000080115cc00@5f455d91


scala> val plus1 = HOModule.partial(1, plus)
val plus1: Int => Int = HOModule$$$Lambda$1710/0x000000080115d7c8@1368ed98


scala> plus1(5)
val res11: Int = 6
```

Currying (and uncurrying)

Currying

Currying is converting a function that takes multiple arguments into a sequence of functions that each take one argument.

If we have a function f and $x = f(a,b)$, then currying would result in two functions g and h such that $g(a) = h$ and $h(b) \ = \ f(a,b)$. In other words:

$$f(a,b) = g(a)(b)$$

Uncurrying

Uncurrying is the inverse of currying.

Uncurrying takes as input a function f that returns a new function g and returns a function h that takes as input the parameters for f and g.

Currying in Scala

```
def mul(x: Int, y: Int) = x * y

def cmul(x: Int)(y: Int) = x * y


----------


scala> mul(3,6)
val res0: Int = 18


scala> val threetimes = cmul(3)
val threetimes: Int => Int =
Lambda$1575/0x0000000801178a00@736a8aea


scala> threetimes(9)
val res1: Int = 27
```

Variables are bound left to right. Until the last variable, each binding of a variable returns a new function.

Functional Data Structures

Functional Data Structures

- In this part of the course, we will implement our own functional data structures in order to gain a better understanding of the concept.
- Implementing our own functional data structures furthermore allows us to learn concepts such as pattern matching.
  - You will, of course, use the functional data structures provided by Scala for your projects and tests.

- In functional programming, we are not allowed to update variables or use mutable data structures. In functional programming, we use functional data structures that are immutable.
  - For example, concatenating two lists `l1` and `l2` yields a new list instead of changing `l1`.

- We have already seen (and used) immutable lists from Scala's standard library. We will now implement our own version of List.

Algebraic Data Types – ADTs

- An Algebraic Data Type (ADT) is a composite type that is formed by combining other types. An ADT prescribes the shape of its composition and constituents.
- Why is it called algebraic? Because we describe ADTs in terms of their sum and product. An ADT is the
  - Sum of their data constructors, and
  - Product of its arguments

- Singly linked lists is a common ADT
  - The sum has two variants:
    - `Nil` for the empty list
    - `Cons(x,l)` for creating a new list from an element `x` and list `l`
  - The product of `Nil` is empty
  - The product of `Cons` is an object and a List.

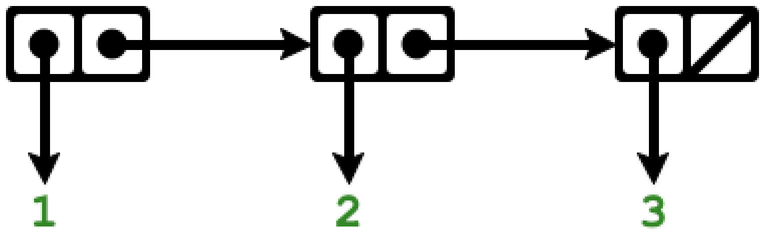
- The product and sum of an ADT allows us to formalize patterns allowing us to operate its elements.

Singly Linked Lists

- cons is a fundamental function from the Lisp programming language and available in most of its dialects (Scheme, Racket,…). It allows us to construct an object that contains two values or pointers to two values. We call these objects cons-cells or cons-pairs.
- In Lisp and its dialects, we can create cons-pairs of any two objects. In Scala, however, cons-pairs are restricted to the creation of singly linked lists.
- In Scala, the cons operator is `::`
  - The first element of a cons-pair, called the head, can contain (a reference to) any value.
  - The second element of a cons-pair, called the tail, must be a singly linked list or the empty list.

```
scala> List(1, 2, 3)
val res0: List[Int] = List(1, 2, 3)
```

```
scala> 1 :: 2 :: 3 :: Nil
val res1: List[Int] = List(1, 2, 3)
```



`List(1, 2, 3)` and `1 :: 2 :: 3 :: Nil` are equivalent.

In the next few slides, we will create our own implementation of lists.

Our own singly Linked Lists

```
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

object List:
    // Companion object
```

- We introduce a new datatype with the `enum` keyword.
- An `enum` allows us to declare a data type that has one or more data constructors that are defined by the `case` keyword. We have two data constructors for `List`:
    - `Nil` represents the empty list, and
    - `Cons` represents a cons-pair where the head is an object and the tail a list.
- Notice that lists are defined in an inductive manner; the base case is the empty list, and all lists are constructed by adding elements to the empty list.

Our own singly Linked Lists

```scala
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

object List:
    // Companion object
```

- The list data type is polymorphic. What does the + in from of the type parameter mean?
- That means that A is a covariant parameter of List.
    - Assuming that `Cat` is a subclass of `Animal`, then with `List[+A]` all `List[Cat]` are also a `List[Animal]`. I.e., `List` varies together with all specializations of its type parameter.
- This is important as we do not want to create a representation of empty list for every type of object.
- Import: Scala infers that `Nil` corresponds with `List[Nothing]`. The keyword `Nothing` is the subtype of all classes and `Nil` can therefore be considered as a `List[Int]`, `List[String]`, `List[Cat]`, …

Variances

- If a type parameter has no `+` or `-`, then the type parameter is invariant.
  - Assuming that `Cat` is a subclass of `Animal`, then with `List[A]` a `List[Cat]` is not a `List[Animal]`. I.e., it does not vary with all specializations of its type parameter.

- If a type parameter has a `-` in front of it, then the type parameter is contravariant.
  - Assuming that `Cat` is a subclass of `Animal`, then with `List[-A]` all `List[Animal]` are also a `List[Cat]`.
  - Is this useful? Yes. However, this will make much more sense after a course on Object-Oriented Programming in the next semester.
  - If you are interested, you can read more about it in https://docs.scala-lang.org/tour/variances.html

Our own singly Linked Lists

```
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

object List:
    // Companion object
```

- A companion object is an object that has the same name as our data type.
- The companion object will allow us to access convenience functions for creating, managing, working, … with objects of that data type.
- Is the provision of a companion object for a data object mandatory? No, but it is common and a best practice.

Data type constructors with variadic functions

Let's create a data type constructor with variadic functions. Variadic functions are functions that accept zero or more arguments.

`as: A*` means `as` is a sequence (`Seq`) object of elements of type `A`.

A sequence is an ordered list. A sequence has functions such as `isEmpty`, `head`, and `tail`. Be careful, they are not the same as `head` and `tail` of our `List`.

```
object List:
    // ...

def apply[A](as: A*): List[A] =
    if(as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
```

```
----------
scala> :load List.scala

scala> import List._

scala> val x = Cons(1, Cons(2, Cons(3, Nil)))
val x: List[Int] = Cons(1,Cons(2,Cons(3,Nil)))

scala> val y = List(1, 2, 3)
val y: List[Int] = Cons(1,Cons(2,Cons(3,Nil)))
```

Data type constructors with variadic functions

We use the apply function, which allows us to create lists with `List(1, 2, 3)`.

Notice how we recursively traverse the Seq object and use our Cons to create the list.

Important detail: `as.tail` returns a Seq object. But apply is a variadic function and we need to treat `as.tail` as arguments (and not one argument). We therefore use use Scala's `_*` operator to adapt a sequence so it can be used as a variable number of arguments.

```
object List:
    // ...

def apply[A](as: A*): List[A] =
    if(as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
```

Variable arguments, or *varargs*.

Creating nested Cons objects via this recursive call.

```
----------

scala> val x = Cons(1, Cons(2, Cons(3, Nil)))
val x: List[Int] = Cons(1,Cons(2,Cons(3,Nil)))


scala> val y = List(1, 2, 3)
val y: List[Int] = Cons(1,Cons(2,Cons(3,Nil)))
```

Pattern matching

Let's create convenience functions in our companion object with pattern matching.

Pattern matching allows us to check a value against a pattern. The pattern with which we match may furthermore deconstruct a value in its constituent parts.

The expression on the left-hand side of the `match` operator is the target that we want to scrutinize.

The right-hand side of the `match` operator contains one or more cases wrapped between curly braces.

```
object List:
    // ...

def product(doubles: List[Double]): Double =
    doubles match
        case Nil => 1
        case Cons(0.0, _) => 0.0
        case Cons(f, r) => f * product(r)



----------


scala> product(List(1, 2, 3))
val res1: Double = 6.0
```

Pattern matching

```
scala> product(Cons(1, Cons(2, Cons(3, Nil))))
• Does the list (1, 2, 3) match Nil? No
• Does the list (1, 2, 3) match Cons(0.0, _)? No
• Does the list (1, 2, 3) match Cons(f, r)? Yes, 1 * product(r)
    • Does the list (2, 3) match Nil? No
    • Does the list (2, 3) match Cons(0.0, _)? No
    • Does the list (2, 3) match Cons(f, r)? Yes, 2 * product(r)
        • Does the list (3) match Nil? No
        • Does the list (3) match Cons(0.0, _)? No
        • Does the list (3) match Cons(f, r)? Yes, 3 * product(r)
            • Does the list () match Nil? Yes, return 1
        • 3 * 1
    • 2 * 3
• 1 * 6
6
```

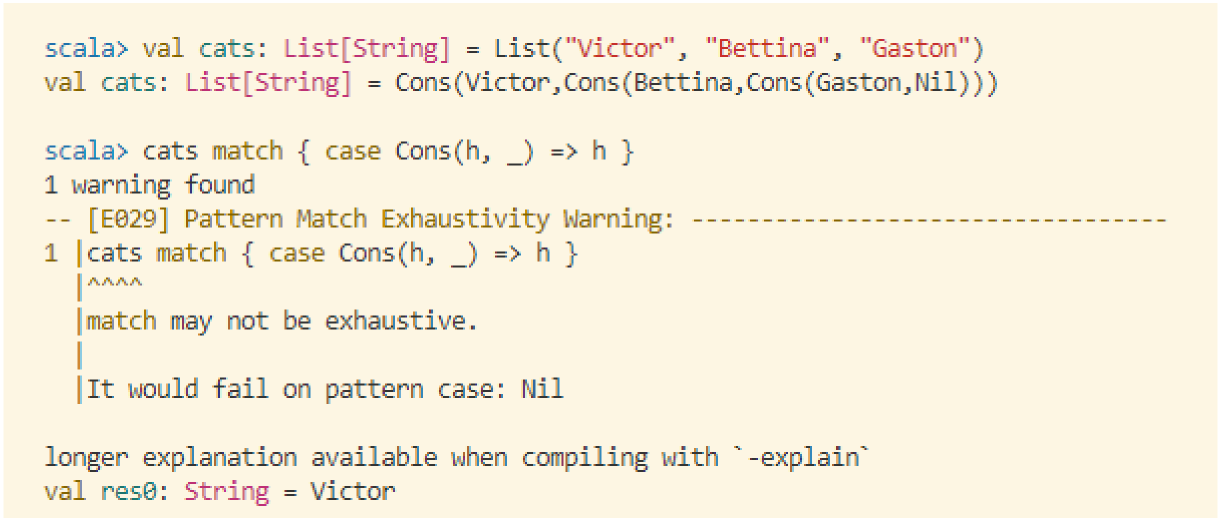
```scala
def product(doubles: List[Double]): Double =
  doubles match
    case Nil => 1
    case Cons(0.0, _) => 0.0
    case Cons(f, r) => f * product(r)
```

Pattern matching: examples

```
scala> val cats: List[String] = List("Victor", "Bettina", "Gaston")
val cats: List[String] = Cons(Victor,Cons(Bettina,Cons(Gaston,Nil)))

scala> cats match { case _ => 42 }
val res0: Int = 42

scala> cats match { case Cons(h, _) => h }
val res1: String = Victor

scala> cats match { case Cons(_, Cons(_, t)) => t }
val res2: List[String] = Cons(Gaston,Nil)

scala> cats match { case Cons("Louis", t) => t }
scala.MatchError: Cons(Victor,Cons(Bettina,Cons(Gaston,Nil))) (of class
rs$line$2$Cons)
```

Pattern matching

- The Scala interpreter may give you warnings when your cases are not exhaustive. I.e., they do not cover all cases. The Scala interpreter may even give you examples of when it would go wrong, leading to a `scala.MatchError`.

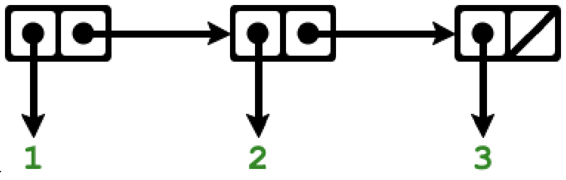


```
scala> val cats: List[String] = List("Victor", "Bettina", "Gaston")
val cats: List[String] = Cons(Victor,Cons(Bettina,Cons(Gaston,Nil)))

scala> cats match { case Cons(h, _) => h }
1 warning found
-- [E029] Pattern Match Exhaustivity Warning: ----------------------------------
1 |cats match { case Cons(h, _) => h }
  |^^^^
  |match may not be exhaustive.
  |
  |It would fail on pattern case: Nil

longer explanation available when compiling with `-explain`
val res0: String = Victor
```

- *You may ignore these warnings <u>if you know what you are doing</u>. :-)*

Data sharing

- How do we manipulate lists (adding elements, removing elements, …)?
  - E.g., `x: List[Int] = List(1, 2, 3)`
  - Adding 4 in front of `x`?
    - We `Cons(4, x)`! Notice that we reuse `x`.
  - Removing the first element?
    - We take `x` and return its `tail`.
    - Notice that we reuse `List(2, 3)` returned by the `tail`.
    - We do not remove things.



- Since lists are immutable, we do not need to create copies but reuse them. This is called data sharing.
- Functional data structures are persistent, which means they are never changed by operations.
- Immutable data structures are always safe to share (as they are not allowed to be changed). With FP, we can therefore achieve greater efficiency w.r.t. memory usage compared to approaches with side effects where one may need to make copies of data structures.

Example of data sharing

```
    def append[A](l1: List[A], l2: List[A]): List[A] = l1 match
        case Nil => l2
        case Cons(h, t) => Cons(h, append(t, l2))

----------


scala> val x = List(1, 2)
val x: List[Int] = Cons(1,Cons(2,Nil))

scala> val y = List(3, 4)
val y: List[Int] = Cons(3,Cons(4,Nil))

scala> append(x, y)
val res4: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
```

Example of data sharing

```
def append[A](l1: List[A], l2: List[A]): List[A] = l1 match
    case Nil => l2
    case Cons(h, t) => Cons(h, append(t, l2))
```

Only the values of `l1` are copied into new `Cons` objects. The `l2` is returned when the `l1` is exhausted. Where the tail of the last cons-pair of `l1` would refer to `Nil`, the tail of the new cons-pair will now refer to `l2`.

Compared to arrays, where a new array needs to be created in which the values of both arrays are copied into, appending immutable lists is much more efficient.

Are all functions much more efficient with immutable lists? No.

allButLast

```
    def allButLast[A](l: List[A]): List[A] = l match
        case Nil => Nil // We can argue over this :-)
        case Cons(_, Nil) => Nil
        case Cons(h, t) => Cons(h, allButLast(t))
```

We will handle errors and exceptions later.

```
----------

scala> allButLast(List(1,2,3))
val res0: List[Int] = Cons(1,Cons(2,Nil))

scala> allButLast(List(1))
val res1: List[Int] = Nil

scala> allButLast(List())
val res2: List[Nothing] = Nil
```

<u>Why can this function not be implemented in constant time?</u> Whenever we need to change the tail of a cons in a singly linked list (e.g., to retain all but the last element), <u>we must copy all the previous cons-pairs</u>.

There are other immutable data structures that do provide such functions in constant time (e.g., Vector), but we will cover those later in this course.

Generalizing recursion over lists with HOF

```
def product(doubles: List[Double]): Double = doubles match
    case Nil => 1
    case Cons(0.0, _) => 0.0
    case Cons(first, rest) => first * product(rest)

def sum(integers: List[Int]): Int = integers match
    case Nil => 0
    case Cons(first, rest) => first + sum(rest)
```

- Here we have two functions `product` and `sum`.
- Even though `product` has an additional case, <u>they are very similar in structure</u>.
- Can we generalize these two functions? Yes.
    1. Subexpressions become function arguments.
    2. If subexpression uses local variables, the functions passed as function arguments will accept these variables as arguments.

Generalizing recursion over lists with HOF

```
def foldRight[A, B](as: List[A], z: B)(f: (A, B) => B): B = as match
    case Nil => z
    case Cons(h, t) => f(h, foldRight(t, z)(f))

def product(doubles: List[Double]) =
    foldRight(doubles, 1.0)(_ * _)

def sum(integers: List[Int]) =
    foldRight(integers, 0)(_ + _)
```

- What does `foldRight` do?
  - It replaces the cons with a function and applies that function to the cons' head and tail.
  - In `sum(List(1,2,3))`,
    `Cons(1, Cons(2, Cons(3, Nil)))` becomes
    `+    (1, +    (2, +    (3, 0)))`

Tracing the evaluation of `sum(List(1,2,3))`

We replace `sum` with its definition.

```
sum(List(1, 2, 3))
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)(_ + _)
1 + (foldRight(Cons(2, Cons(3, Nil)), 0)(_ + _))
1 + (2 + (foldRight(Cons(3, Nil), 0)(_ + _)))
1 + (2 + (3 + (foldRight(Nil, 0)(_ + _))))
1 + (2 + (3 + (0)))
1 + (2 + (3))
1 + (5)
6
```

We replace `foldRight` with its definition.

Notice how `foldRight` must traverse the whole list and place frames onto the call stack before it can collapse.
Is this implementation of `foldRight` tail-recursive? No!

Lists in Scala's standard library

- Cons-pairs are created with `::`, which is an operator that associates to the right.

```
scala> 1 :: 2 :: Nil
val res0: List[Int] = List(1, 2)
scala> 1 :: (2 :: Nil)
val res1: List[Int] = List(1, 2)
scala> List(1,2)
val res2: List[Int] = List(1, 2)
```

- Scala's list has many useful convenience functions. Many will be covered in the exercise sessions, but it is up to you to consult the documentation to prepare yourself for the project, tests and exam.

- Examples of useful convenience functions include `take`, `takeWhile`, `forall`, `exists`, `scanLeft`, `scanRight`, `map`, `flatMap`, `groupMap`, …

Binary trees

Let's define the ADT Tree for representing binary trees.

```
enum Tree[+A]:
    case Leaf(value: A)
    case Branch(left: Tree[A], right: Tree[A])
```

- The sum of `Tree` is
  - `Leaf` for representing leaves in a tree
  - `Branch` for representing a tree by putting two trees together as a left branch and a right branch
- The product
  - Of `Leaf` is an object
  - Of `Branch` are two trees

ADTs provide us a way to "compute" the pattern that we can match. Let's define a function size counting the number of nodes (branches and leaves) in a tree.

Binary trees

```
enum Tree[+A]:
    case Leaf(value: A)
    case Branch(left: Tree[A], right: Tree[A])

    def size: Int = this match
        case Leaf(_) => 1
        case Branch(l, r) => 1 + l.size + r.size

----------

scala> :load Tree.scala

scala> val t = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))
val t: Branch[Int] = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))

scala> t.size
val res0: Int = 5
```

Pattern match on this (a special keyword for "this object") to determine whether size is getting called on a Leaf or a Branch.

Also notice how this function is declared within the enum.

Tuples

- Tuples (and pairs, which are a specific case of tuples) are very convenient ADTs provided by Scala. We have already used tuples in the first chapter to return pairs of coffees and a charge.

- While tuples have a special syntax …

```
scala> val cats = ("Victor", "Bettina", "Gaston")
val cats: (String, String, String) = (Victor,Bettina,Gaston)

scala> cats._2
val res1: String = Bettina

scala> cats match { case (a, b, c) => b }
val res2: String = Bettina
```

- … tuples are syntactic sugar for specific types of objects. E.g., tuples having the type `(Int, Int, String)` correspond with objects of type `Tuple[Int, Int, String]`.

Traits vs enumerations

- Currently, we have seen examples of enum to represent data types. Another option is to use trait and case classes and object. Traits are used to share interfaces (names of methods, and therefore also functions) and fields between classes.

- In Scala, an enum is exhaustive. A trait, however, can be extended except when they keyword **sealed** is used. When that keyword is used, then all case classes and case objects must be defined in that file.
  - Remember that a case object is a case class with only one object with that same name.

- For this course, the difference is not that important. Some differences are:
  - You can access the parameters of a case class via their names.
  - Case classes are available in the scope whereas the different object of an enum need to be imported.
  - You can iterate over all objects inside an enum. This is not possible with instantiated case classes and case objects.
  - enums are "easier" to persist in files, databases, etc. The persistence of instantiated case classes and case objects needs to be programmed (but there are libraries, of course).

Traits vs enumerations

For instance, with our Tree, we cannot retrieve a branch's left or right:

```
enum Tree[+A]:
    case Leaf(value: A)
    case Branch(left: Tree[A], right: Tree[A])

----------

scala> import Tree._

scala> val t = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))
val t: Branch[Int] = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))

scala> t.left
-- [E006] Not Found Error: ---------------------------------
1 |t.left
  |^^^^^^
  |value left is not a member of Tree[Int] - did you mean t.wait?
1 error found
```

Traits vs enumerations

This is possible with a trait.

```
sealed trait Tree[+A]:
    def size: Int = this match
        case Leaf(_) => 1
        case Branch(l, r) => 1 + l.size + r.size

case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]

----------

scala> val t = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))
val t: Branch[Int] = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))

scala> t.left
val res0: Tree[Int] = Branch(Leaf(1),Leaf(2))
```

Lexicon

- Accumulator – accumulateur
- Algebraic Data Type (ADT) – type algébrique de données
  - != Abstract Data Type (ADT) – Type abstrait
- Arity – arité
- Data structure – structure de données
- Higher-order function – fonction d'ordre supérieur
- Higher-order programming – programmation d'ordre supérieure
- Pattern – motif
- Pattern matching – filtrage par motif
- Polymorphism – polymorphisme
- Variadic function – fonction variadique