

INFO0054-1

Programmation Fonctionnelle

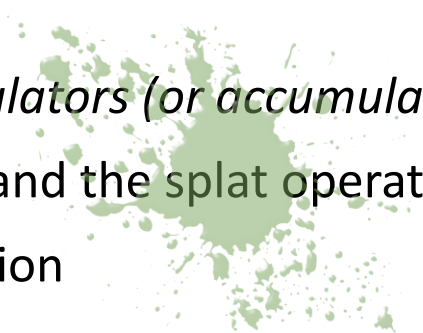
Chapter 03: Recursion

Christophe Debruyne
(c.debruyne@uliege.be)

References

- Not mandatory: Chapter 5: Récursivité structurelle. Pascal Gribomont. Éléments de programmation en Scheme. Cours et exemples d'application. Dunod.
 - *But watch out ! I have provided a little sidenote surrounding the term mixed recursion.*

Overview

- Structural Recursion
 - On natural numbers
 - On lists
 - On nested lists (or trees)
 - Complete structural recursion
 - *Recursion without and with accumulators (or accumulative recursion)*
 - A little detour: variadic arguments and the splat operator
 - Non-structural or generative recursion
- 

Introduction

- So far, we have seen some Algebraic Data Types such as (Natural) Numbers, Tuples, and Lists.
- In this lesson, we will have a closer look at some of these ADTs and define some common approaches to solving problems by using the inductive definition of these ADTs.

Structural recursion

- Scala accepts any syntactically correct recursive function definition, even if the function may never terminate (i.e., an infinite number of computations).

```
def test(n: Int): Int =  
  if(n == 0) 1  
  else test(n + 1) + n
```

- It is up to the developer to know if, for a given domain, the computation will always terminate.
- This verification task can be tedious but, for some ADTS, **there exists "templates" that guarantee termination**. The most useful of these schemes are **structural templates**, based on the way objects in the are composed or constructed.
- With structural templates, the evaluation process reduces the computation of $f(v)$ to the computation of $f(v_1), \dots, f(v_n)$ where the v_i are the **direct components** of v .
 - This technique is safe if we limit ourselves to the domains whose objects have a finite number of clearly defined components (direct or not).

Structural recursion on natural numbers

- Conceptually, the natural numbers are built from 0 and the *successor* function.
 - 0 has no component; it is the **base object**.
 - Any other natural number n has a direct component, i.e., its **predecessor**.
- For natural numbers, structural recursion consists in
 - Defining $f(0)$ as a constant, and
 - The "constant" may be an expression using arguments, but may not contain recursive calls.
 - Defining $f(n)$, with $n > 0$, as the value of an expression involving a recursive call to $f(n - 1)$. No recursive calls to indirect components of n are permitted.
- We can define functions with several arguments, but the recursion stands on only one of these arguments.
 - I.e., if you have 3 arguments, then the recursion is only defined on one of these three arguments.

Example

```
def times(n: Int, m: Int): Int =  
    if(n == 0) 0  
    else m + times(n - 1, m)
```

- When $n = 0$, then we return a constant.
 - We call this the **base case**.
- When $n > 0$, then we have an expression with a recursive call using $n - 1$.
- Now let's "rewrite" this function so that the structural template becomes apparent.

Example

```
def times(n: Int, m: Int): Int =  
  def F(n: Int, u1: Int): Int =  
    if(n == 0) G(u1)  
    else H(F(n - 1, K1(n, u1)), n, u1)  
  
  def G(u1: Int) = 0  
  def K1(n: Int, u1: Int) = u1  
  def H(vrec: Int, n: Int, u1: Int) =  
    u1 + vrec  
  
  F(n, m)
```

We have multiple arguments. This recursion stands on n and we have one additional argument.

Example

```
def times(n: Int, m: Int): Int =  
  def F(n: Int, u1: Int): Int =  
    if(n == 0) G(u1)  
    else H(F(n - 1, K1(n, u1)), n, u1)  
  
  def G(u1: Int) = 0  
  def K1(n: Int, u1: Int) = u1  
  def H(vrec: Int, n: Int, u1: Int) =  
    u1 + vrec  
  
  F(n, m)
```

The base case (n is equal to 0):

- The function **G** takes as input all the other arguments to return the constant.
- In this case, **u1** is 'ignored' and **0** is returned.

There are instances of problems where the arguments are used or returned. E.g., an accumulator.

Example

```
def times(n: Int, m: Int): Int =  
  def F(n: Int, u1: Int): Int =  
    if(n == 0) G(u1)  
    else H(F(n - 1, K1(n, u1)), n, u1)  
  
  def G(u1: Int) = 0  
  def K1(n: Int, u1: Int) = u1  
  def H(vrec: Int, n: Int, u1: Int) =  
    u1 + vrec  
  
  F(n, m)
```

The other cases:

- The function `H` takes as input: the value of the recursive call, `n`, and the other args.
- In this example, `H` adds the value of `u1` to the value of the recursive call `vrec`.

Example

```
def times(n: Int, m: Int): Int =  
  def F(n: Int, u1: Int): Int =  
    if(n == 0) G(u1)  
    else H(F(n - 1, K1(n, u1)), n, u1)
```

```
  def G(u1: Int) = 0
```

```
  def K1(n: Int, u1: Int) = u1
```

```
  def H(vrec: Int, n: Int, u1: Int) =  
    u1 + vrec
```

```
  F(n, m)
```

- The recursive call provides for each additional argument `ux` a function `K1` to manipulate the variable based on all `ux`. In this case, `K1` just returns `u1`.
- If we had 2 additional variables, we would have had:
 - `K1(n, u1, u2)` to compute a new value for `u1`.
 - `K2(n, u1, u2)` to compute a new value for `u2`.
- These functions can be defined to "update" the values of our accumulators.

Example 2

When there are no additional arguments, then the template becomes simple:

```
def F(n: Int): Int =  
    if(n == 0) G()  
    else H(F(n - 1), n)  
  
def G() = ...  
  
def H(vrec: Int, n: Int) = ...
```

```
def fac(n: Int): Int =  
    def F(n: Int): Int =  
        if(n == 0) G()  
        else H(F(n - 1), n)  
    def G() = 1  
    def H(vrec: Int, n: Int) =  
        n * vrec  
    F(n)
```

- Generally, we will not write functional programs in such a way. The function `F` is not tail-recursive, for starters.
- But (!), these structural templates provide a guideline or framework to think about a problem. A programmer only needs to define the functions `G`, `H`, and `K1`, ..., `Km`.

By the way

- You can revise this structural template (and others) with higher-order programming to provide the functions **G**, **H**, and **K1**, ..., **Km** as arguments to **G**.
- Try this at home to become more familiar with higher-order programming!

Structural recursion on lists

- Scala lists are similar to arrays, but:
 - Lists are immutable, i.e., the elements of a list cannot be changed by assignment;
 - Lists are linked lists (cons-pairs referring to one another) whereas arrays are a “flat” data structure
- The following two expressions return the same list

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

scala> val y = 1 :: 2 :: 3 :: Nil
val y: List[Int] = List(1, 2, 3)
```
- Remember: `::` is Scala’s cons operator used to prepend a **head** (and element) to a **tail** (the remainder of a list). All lists should end with `Nil`.
- We thus start building lists from `Nil` by prepending elements. `Nil` is thus our base case.

Flat recursion

- The way lists are defined and built in an inductive manner leads us to the following template for flat recursion.
- Let's start with an example where we count the elements of a list satisfying a condition.

```
def countif[A](l: List[A], f: A => Boolean): Int =  
  if(l.isEmpty) 0  
  else (if f(l.head) then 1 else 0) + countif(l.tail, f)
```

- Now let's "rewrite" this function so that the structural template for flat recursion becomes apparent.

Example

```
def countif[A](l: List[A], f: A => Boolean): Int =
```

```
  def F[A](l: List[A], u1: A => Boolean): Int =  
    if(l.isEmpty) G(u1)  
    else H(F(l.tail, K1(l, u1)), l, u1)
```

Again, the function
F takes care of
the recursion.

```
  def G[A](u1: A => Boolean) = 0  
  def K1[A](l: List[A], u1: A => Boolean) = u1  
  def H[A](vrec: Int, l: List[A], u1: A => Boolean) =  
    (if u1(l.head) then 1 else 0) + vrec
```

And these
functions are to
be implemented
by the programmer.

```
F(l, f)
```


Example

```
def countif[A](l: List[A], f: A => Boolean): Int =  
  def F[A](l: List[A], u1: A => Boolean): Int =  
    if(l.isEmpty) G(u1)  
    else H(F(l.tail, K1(l, u1)), l, u1)  
  
  def G[A](u1: A => Boolean) = 0  
  def K1[A](l: List[A], u1: A => Boolean) = u1  
  def H[A](vrec: Int, l: List[A], u1: A => Boolean) =  
    (if u1(l.head) then 1 else 0) + vrec
```

`F(l, f)`

This implements the base case.

A recursive call on the remainder of the list.

We use the “current” list rather than the head to make our functions as generic as possible. We will often use the head of the list, however.

Example 2

When there are no additional arguments, then the template becomes simple:

```
def F[A](l: List[A]): Int =  
  if(l.isEmpty) G()  
  else H(F(l.tail), 1)
```

```
def G[A]() = ...
```

```
def H[A](vrec: Int, l: List[A]) =  
  ...
```

```
def count[A](l: List[A]): Int =  
  def F[A](l: List[A]): Int =  
    if(l.isEmpty) G()  
    else H(F(l.tail), 1)  
  def G[A]() = 0  
  def H[A](vrec: Int, l: List[A]) =  
    vrec + 1  
  F(l)
```

Some more examples

- We can use these templates to implement the functions `G`, `H`, and `K1`, ..., `Km`.
- In practice, however, they provide a guideline and framework. We can use these templates to write down some functions that rely on flat recursion.

```
def length[A](l: List[A]): Int =  
  if(l.isEmpty) 0  
  else 1 + length(l.tail)
```

```
def append[A](l1: List[A], l2: List[A]): List[A] =  
  if(l1.isEmpty) l2  
  else l1.head :: append(l1.tail, l2)
```

```
def reverse[A](l: List[A]): List[A] =  
  if(l.isEmpty) Nil  
  else append(reverse(l.tail), List(l.head))
```

Notice how `append` and `reverse` “return” lists that we have built.

Flat recursion revisited

- The template we have seen for flat recursion on lists uses an if-statement.
- We have seen that purely functional data structures and pattern matching go well together.
- We will revisit an example and our template to use pattern matching instead.

```
def countifrev[A](l: List[A], f: A => Boolean): Int =  
  def F[A](l: List[A], u1: A => Boolean): Int = l match  
    case Nil => G(u1)  
    case _ => H(F(l.tail, K1(l, u1)), l, u1)  
  
  def G[A](u1: A => Boolean) = 0  
  def K1[A](l: List[A], u1: A => Boolean) = u1  
  def H[A](vrec: Int, l: List[A], u1: A => Boolean) =  
    (if u1(l.head) then 1 else 0) + vrec  
  
  F(l, f)
```

- You will notice that not that much has changed. One may argue that pattern matching is much more elegant.

Flat recursion revisited II

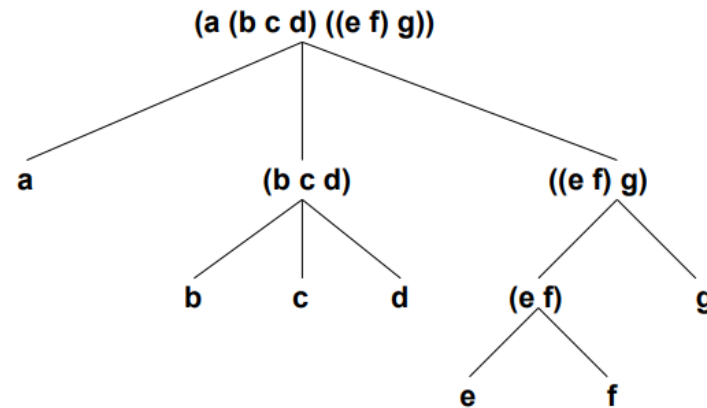
- Here's a slight variation of our function with pattern matching. Instead of a "catch all", we take advantage of the way lists are built.

```
def countifrev[A](l: List[A], f: A => Boolean): Int =  
  def F[A](l: List[A], u1: A => Boolean): Int = l match  
    case Nil => G(u1)  
    case h :: t => H(F(t, K1(l, u1)), l, u1)  
  
  def G[A](u1: A => Boolean) = 0  
  def K1[A](l: List[A], u1: A => Boolean) = u1  
  def H[A](vrec: Int, l: List[A], u1: A => Boolean) =  
    (if u1(l.head) then 1 else 0) + vrec  
  
  F(l, f)
```

- Knowledge of these variations will come in handy in writing code that is easy to read and comprehend, which of course depends on the problem you try to solve.

Deep recursion

- Flat recursion only considers the elements of a list, which are all the same type (`Int`, `String`, `Any`, `List`, ...).
- We can nest lists in order to create trees. For instance:
`val t = List("a", List("b", "c", "d"), List(List("e", "f"), "g"))`
corresponds with the tree

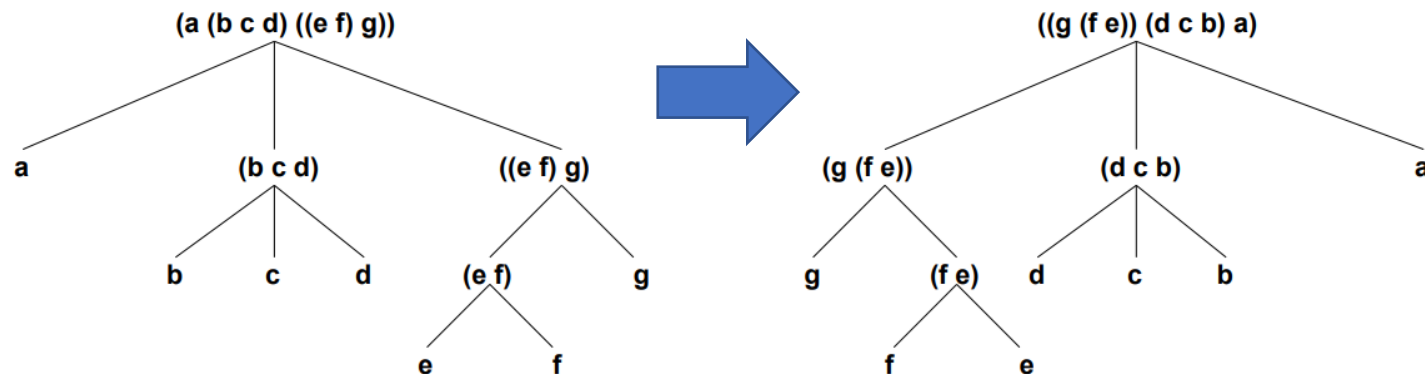


- What happens when we reverse this tree?

Deep recursion

```
scala> reverse(t)  
val res0: List[Object] = List(List(List(e, f), g), List(b, c, d), a)
```

- Only the elements of the “top” list are reversed!
- What if we wanted `List(List(g, List(f, e)), List(d, c, b), a)`



- Is deep recursion a different type of structural recursion?
- Yes, and it has its own template.

Deep recursion

- The conditions of flat recursion were as follows:
 1. If the list is empty -> *do something*
 2. Else (i.e., the list not empty) -> *do something*
+ recursive call with tail
- For flat recursion, we must add an additional condition:
 1. If the list is empty -> *do something*
 2. If head is atomic -> *do something*
+ recursive call with tail
 3. Else (i.e., head is list) -> *do something*
+ recursive call with head
+ recursive call with tail
- Before we demonstrate the template...

Scala's typing system

- Even though Scala has an advanced typing system, ...
 - (cf. type inference mechanisms)
- Scala is strongly and statically typed.
 - Strong means that variables must have a type.
 - Static means that the type of variables cannot change over time.
- Why bring this up? There are programming languages with weak and dynamic typing that makes functional programming much, much easier.
 - But languages such as Scala are currently more "popular" in industry.

- Let's look at an example in Scheme / Racket:

```
(define deeprev
  (lambda (l)
    (cond ((empty? l) '())
          ((atom? l) l)
          (else (append (deeprev (cdr l))
                        (list (deeprev (car l)))))))
```

Notice that the variable `l` can be either a list or something else. The absence of strong typing makes our pattern much more visible.

- In Scala, we either must work with abstractions (see later chapters) or with pattern matching.

Example

```
def deepRev(l: List[_]): List[_] = l match
  case Nil => Nil
  case (h: List[_]) :: t => append(deepRev(t), List(deepRev(h)))
  case (h: Any) :: t => append(deepRev(t), List(h))
```

```
scala> val t2 = List(1, List(2, 3, 4), List(List(5, 6), 7))
val t2: List[Matchable] = List(1, List(2, 3, 4), List(List(5, 6), 7))
```

```
scala> deepRev(t2)
val res0: List[?] = List(List(7, List(6, 5)), List(4, 3, 2), 1)
```

- Notice how "cumbersome" managing the types and signatures is.
- Also notice the order of the cases. Here, we must be careful with the order as the third is more general than the second.
- If you prefer keeping the pattern *empty?-atom?-else*, then you need to work with [guarded pattern matching](#). Guards are condition which follow the pattern with an "if" keyword allowing us to formulate specific conditions of similar patterns. We will not cover these (much) in class.

Example 2

```
def scale(l: List[_], n: Int): List[_] = l match
  case Nil => Nil
  case (h: List[_]) :: t => scale(h, n) :: scale(t, n)
  case (h: Int) :: t => (h * n) :: scale(t, n)
```

```
scala> val t2 = List(1, List(2, 3, 4), List(List(5, 6), 7))
val t2: List[Matchable] = List(1, List(2, 3, 4), List(List(5, 6), 7))

scala> scale(t2, 5)
val res2: List[?] = List(5, List(10, 15, 20), List(List(25, 30), 35))
```

- A pattern is emerging... We can create abstractions for this pattern like we did for the previous data structures.

Example 2

```
def scale2(l: List[_], n: Int): List[_] =  
  def F(l: List[_], u1: Int): List[_] = l match  
    case Nil => G(u1)  
    case (h: List[_]) :: t => J( F(h, A1(l, u1)), F(t, D1(l, u1)), l, u1)  
    case (h: Int) :: t => H( F(t, K1(l, u1)), l, u1 )  
  
  def G(u1: Int) = Nil  
  def K1(l: List[_], u1: Int) = u1  
  def A1(l: List[_], u1: Int) = u1  
  def D1(l: List[_], u1: Int) = u1  
  def H(vrec: List[_], l: List[_], u1: Int) =  
    l.head.asInstanceOf[Int] * u1 :: vrec  
  def J(vrech: List[_], vrect: List[_], l: List[_], u1: Int) =  
    vrech :: vrect  
  
  F(l, n)
```

Summary (so far)

- We have seen templates for natural numbers, lists, and nested lists.
- You can adopt these templates, which provide you a methodological approach to solving a problem. It furthermore has the guarantee to terminate, especially when you restrict yourselves to implementing the auxiliary functions H, K, etc.
- However, you can (and should) use these templates as a framework. I.e., you should be inspired by those templates and not necessarily stick to the exact functions. There are many reasons for doing that, e.g.:
 - Replace G by a constant if it only returns a constant value;
 - Avoid the use of expensive functions (e.g., append);
 - Ensure that the function uses tail-recursion;
 - ...

A small “detour”: Using variadic arguments

Variadic arguments

- A variadic function is a function with an indefinite arity. In other words, it accepts a variable number of arguments (zero, one, or more).
- In Scala (and other programming languages), we can allow a function to accept any number of (extra*) arguments.
 - * depending on the signature
- These arguments are known as variadic arguments, variable arguments, varargs, or **repeated arguments**.
- **Repeated arguments** is the term used in the Scala community, but the others are valid as well. In short, a repeated argument of type **A** allows us to treat an argument as an **ArraySeq[A]**.
- A function can have at most one repeated parameter, which must be the last parameter, and all elements must have the same type.

Variadic arguments

- Example:

```
def sum(args: Int*): Int = args.fold(0)(_+_)
```

```
scala> sum(3, 6, 9)
val res0: Int = 18
```

```
scala> sum()
val res1: Int = 0
```

- `*` means that if 0, 1, or more arguments of type `Int` are provided in the function call, all these values will be stored in a `ArraySeq[Int]`.
- `ArraySeq[A]` are an immutable data structure like `List[A]` for which provides support for well known functions such a `map`, `fold`, `isEmpty`, etc.
- We can easily adapt our templates to variadic arguments.

The splat operator

- Given

```
def sum(args: Int*): Int = args.fold(0)(_+_)
```

- How can we provide a `List[Int]`, `Array[Int]`, ... to `sum`?
- We can “splat” such a data structure in the function call with the splat operator.

```
scala> sum(Array(1, 2, 3, 4, 5): _*)  
val res11: Int = 15
```

- The splat operator converts the collection into objects accepted by the repeated parameter.

Let's get back on track! :-)

Complete structural recursion

- The templates we have seen so far follow the inductive definition of our data structures. In those templates, $f(x, \dots)$ is defined in terms of $f(y, \dots)$ where y is a direct component of x .
- We can generalize this approach of structural recursion by allowing:
 - Indirect components of x (e.g., using $n-2, \frac{n}{2}, \dots$ instead of $n-1$)
 - Allowing multiple recursive steps
 - Allowing multiple recursive calls in one step
 - Allowing nested calls
- This generalized approach is called **complete structural recursion**.

Complete structural recursion

- When complete structural recursion is correctly applied, the algorithm is guaranteed to terminate, but not necessarily efficient.
- Since it is not patterned after a template, testing also becomes more challenging as we
 - Need to ensure we have covered all cases.
 - Need to avoid errors (e.g., testing for $n == 0$ and a recursive step with $n - 2$)
 - ...

Complete structural recursion

- Examples of complete structural recursion

```
def exp(m: Int, n: Int): Int =  
    if n == 0 then 1  
    else if n % 2 == 0 then exp(m * m, n / 2)  
    else m * exp(m, n - 1)
```

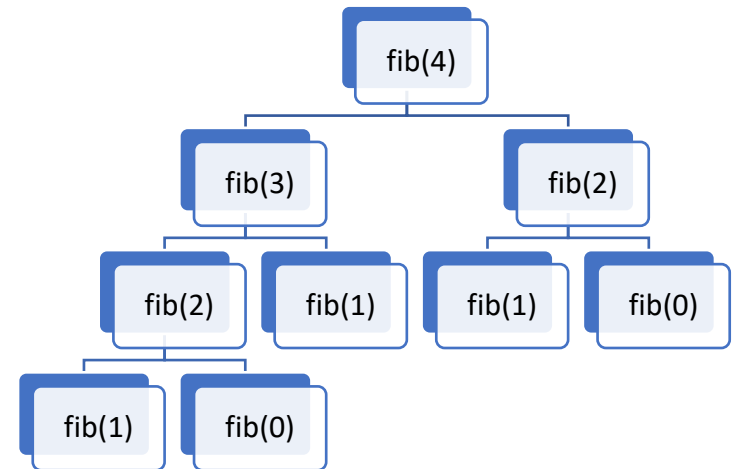
- We have two recursive steps.
- We have recursion on indirect components of n.
- This is efficient. Why?

Complete structural recursion

- Examples of complete structural recursion

```
def fib(n: Int): Int =  
    if n < 2 then n  
    else fib(n - 1) + fib(n - 2)
```

- Base case and other specific cases merged into one.
- One recursive case with two recursive calls.
- This is inefficient. Why?
- This "naïve" definition of `fib` is also tree-recursive as there are multiple recursive calls in a recursive statement.



Fib revisited

Functions, such as the naïve implementation of `fib`, can be redefined in such a way that they avail of the template of structural recursion. Not only that, `fib` can also be defined in such a way that it becomes tail recursive.

```
def fib(n: Int): Int =  
  if n < 2 then n  
  else fib(n - 1) + fib(n - 2)  
  
def fib2(n: Int): Int =  
  @annotation.tailrec  
  def helper(n: Int, a: Int, b: Int): Int =  
    if(n == 0) then a  
    else helper(n - 1, b, a + b)  
  helper(n, 0, 1)
```

The `helper` function uses the template and is tail recursive. Notice how `b` is an accumulator.

Here is a key takeaway from this lesson: we can have functions that are mathematically equivalent, but not computationally equivalent. Here, `fib` and `fib2` are equivalent from a mathematical perspective, but `fib2` is much more efficient!

Non-structural recursion

- Non-structural recursion is also known as *generative recursion*.
- In non-structural recursion, we decompose a problem into subproblems that are neither direct nor indirect components of our data structure on which the function recurses.
- Since there is no template, we need to ensure all cases are covered and tested.
 - In structural recursion, termination is easily shown using the templates.
 - In non-structural recursion, we need to provide proof of termination.
- Not all problems can be solved with structural recursion, so non-structural recursion can help us solve more problems.
- Some problems that are defined with non-structural recursion may be defined in terms of structural recursion, however.
 - We will see an example. :-)

Non-structural recursion

The greatest common divisor, or gcd, of two natural nonzero numbers is the largest natural number that divides both numbers.

```
def gcd(a: Int, b: Int): Int =  
  if a == 0 then b  
  else if b == 0 then a  
  else gcd(b, a % b)
```

We still have a base case(s), recursive steps, and recursive calls, but ... the recursive call does not depend on (in)direct components the data.

In this example, we do not depend on (in)direct components of a and b. Instead, it depends on operations involving a and b.

A little exercise :-)

- Our definition of `gcd` is non-structural. Can we come up with a definition of `gcd` that uses (complete) structural recursion?
- Yes!

```
def gcd2(a: Int, b: Int): Int =  
  def helper(n: Int, a: Int, b: Int): Int =  
    if a % n == 0 && b % n == 0 then n  
    else helper(n - 1, a, b)  
  helper(a, a, b)
```

In this example, `helper` does use the structure of the data; namely the predecessor of `n`.

Recursion on multiple parameters

- Even though we have seen (complete) structural recursive functions taking one or more parameters, we have seen functions that recurse on one parameter.
- Can we have (complete) structural recursive functions that recurse on more than one parameter?
- Yes!
- When such a function has multiple parameters to recurse on, we must ensure that our base cases are covered and that our recursive statements apply recursion on the appropriate (in)direct components of our parameters.
- Examples include zipping lists, computing values in Pascal's triangle, etc.

Recursion on multiple parameters

```
def zip[A, B](l1: List[A], l2: List[B]): List[(A, B)] = (l1, l2) match
  case (Nil, _) => Nil
  case (_, Nil) => Nil
  case (h1 :: t1, h2 :: t2) => (h1, h2) :: zip(t1, t2)
```

Base cases

Breaking down the lists by
taking the head and the tail.

```
scala> zip(List(1,2,3,4),List("Victor","Bettina","Gaston"))
val res0: List[(Int, String)] = List((1,Victor), (2,Bettina), (3,Gaston))
```

By the way

- Recursion on multiple parameters is sometimes called mixed recursion.
- However, you will find that others use mixed recursion for recursion on multiple parameters, regardless whether the subproblems are structural or not. In their interpretation, gcd would also be considered an instance of mixed recursion.
- There's no right or wrong, this depends on how an author defines these terms.
- In this course, however, I will refer to mixed recursion as (complete) structural recursion on multiple arguments.

To conclude

- We have covered
 - Structural Recursion
 - Complete structural recursion
 - Non-structural or generative recursion
- We have seen some examples and we have also seen how we can provide mathematically equivalent, yet computationally very different definitions of functions.
- For (complete) structural recursion, we have covered the templates for common data structures. However, do recall that you can create your own functional data structures (inductive definition) from which your templates "naturally" flow. E.g., our ADT for binary trees.

Lexicon

- Base case – cas de base
- Deep recursion – récursivité profonde
- Flat recursion – récursivité superficielle
- Greatest common divisor – plus grand commun diviseur
- Structural recursion – récursivité structurelle
- Complete structural recursion – récursivité structurelle complète