IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

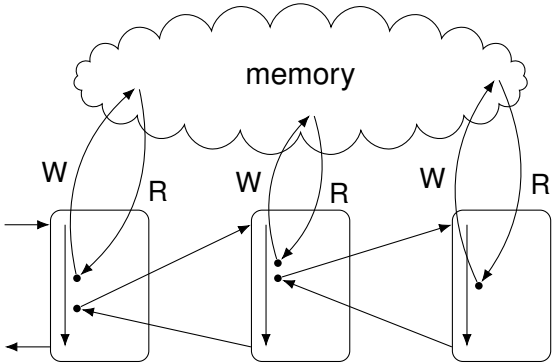
Functional Programming

using Elixir

Fabien Dagnat

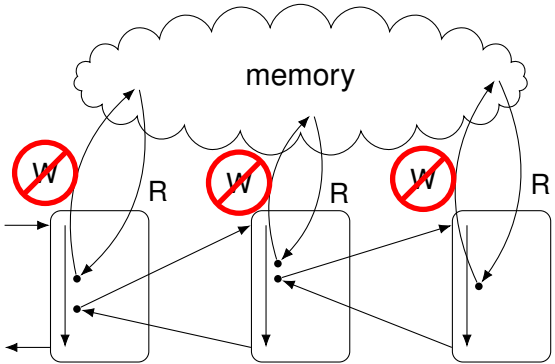
ILSD – Back-to-school week – 2025-2026

An imperative program execution



- functions may use memory through variables
- variables can be read or written
- functions are made of statements
- functions interact by parameters/result and memory
- functions may have side effects (on memory)

A functional program execution



- variables can only be read
- global variables are constants
- functions are expressions
- functions must interact by parameters/result
- functions have no side effects (on memory)

Functional programming

- A program is
  - a set of function definitions
  - expression composing these functions
- A function
  - is an expression
  - takes values as parameters
  - returns a value as a result
  - is a value
- a value is immutable

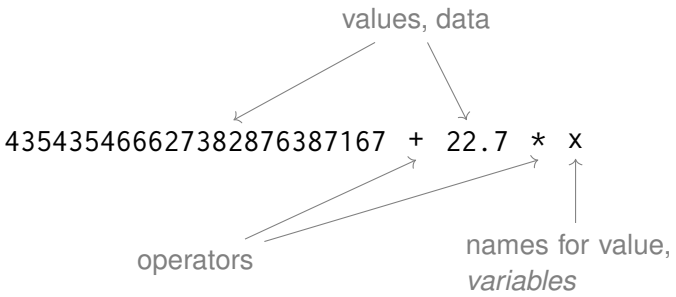
$\Rightarrow$ Approach based on the description of how a value is built from other values

Elixir

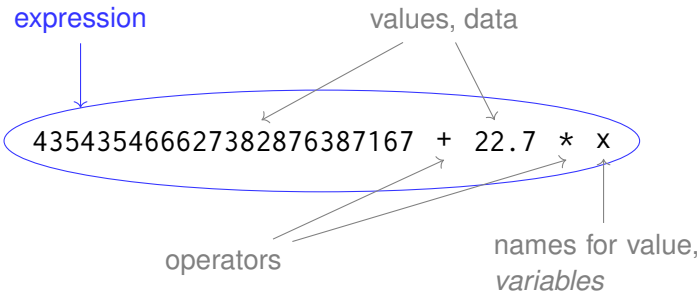
- Functional programming language of the Erlang family
- Proposed by José Valim around 2012
- General principles
  - Functional
  - Concurrent (process executing concurrently)
  - Distributed (notion of node)
  - Dynamic typing (just before execution)
  - Compile to the Erlang Virtual Machine (efficient, distributed, fault tolerant, numerous librairies)

https://p4s.enstb.org/elixir

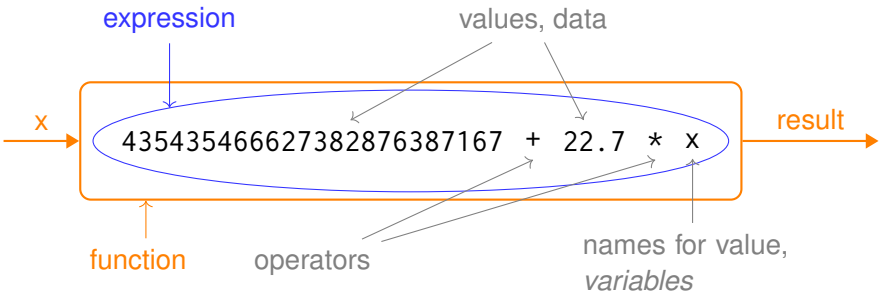
Basic elements



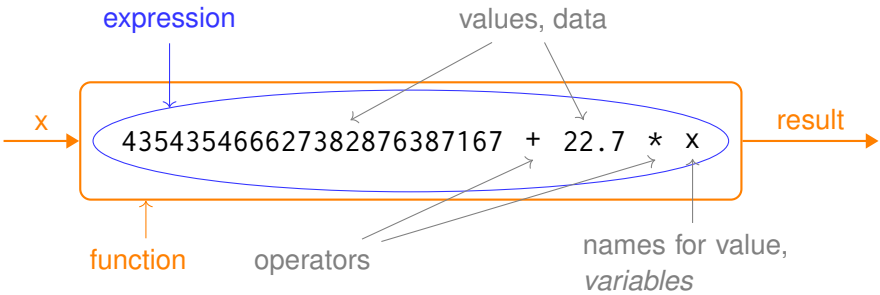
Basic elements



Basic elements



Basic elements



- Variables do not vary!
- Functions are values
  - they can be stored in data structures
  - they can be passer as parameters or returned as results

Semantic by substitution

Variables do not vary

- In imperative languages
  - a variable is memory cell (where we can read and write values)
  - lot of functions act by side effects by modifying (the content of) variables

```
void add(int newVal) { val += newVal; }
```

- In functional languages
  - a variable is a name given to value by a binding

```
my_var = 42
```

  - each use is replaced by the value (substitution)

$\Rightarrow$ a function must return all its results

```
fn (val1,val2) -> val1 + val2 end
```

Functions are values

- They can be stored

```
hello_fun = fn name -> "hello " <> name end
{ "hello function", hello_fun }
```

- We call them by the operator *fun*.(*args*)

`hello_fun.("Fabien")` return the value `"hello Fabien"`

- They can be sent as parameters (high order functions)

```
my_print = fn (fun_msg,name) ->
        IO.puts fun_msg.(name)
      end
```

`my_print.(hello_fun, "Paul")` prints `hello Paul`

Let's practice (on paper)

- The identity function
- A function composing two functions received as parameters
- A function receiving a function `f` and a value `v` and applying `f` to `v`

Let's practice (on paper)

- The identity function

  `fn x -> x end`
- A function composing two functions received as parameters

- A function receiving a function `f` and a value `v` and applying `f` to `v`

Let's practice (on paper)

- The identity function

  ```
  fn x -> x end
  ```

- A function composing two functions received as parameters

  ```
  fn f, g -> fn x -> f.(g.(x)) end end
  ```

- A function receiving a function `f` and a value `v` and applying `f` to `v`

Let's practice (on paper)

- The identity function

  ```
  fn x -> x end
  ```

- A function composing two functions received as parameters

  ```
  fn f, g -> fn x -> f.(g.(x)) end end
  ```

- A function receiving a function `f` and a value `v` and applying `f` to `v`

  ```
  fn f, v -> f.(v) end
  ```

Typing

- Elixir is dynamically typed (before execution)
- in case of typing error, execution is not done and an error is raised for example, `2 + "3"` raises

```
** (ArithmeticError) bad argument in arithmetic expression: 2 + "3"
:erlang.+(2, "3")
```

- writing corrects programs require discipline
- a static type system is under development (since v1.17 of 06/2024)
  `https://elixir-lang.org/blog/2024/06/12/elixir-v1-17-0-released`

Elixir types

- primitives
  - numbers: integers (without limit) and floats (IEEE 754),
  - atoms including **`true`**, **`false`** and **`nil`**
  - identifiers: process, port et reference
- composites
  - ranges (`1..20`)
  - collections: tuples, lists, map and binaries
  - functions

Atom

- An atom is a unique constant value (a symbol different from any other value)
- It has a name (a string)
  - sort of string but built for efficient storage and comparison
- Mostly defined statically using `:mot`
  - word starting by an UTF8 letter and composed of letters, digits, _ and @
  - *e.g.* `:ok` and `:fd@imta`
- Mainly used as names for tags to qualify data
- 3 specific atoms: **`true`**, **`false`** and **`nil`**

Collecting data

- Fixed quantity
  - small quantity ⇒ tuple

    ```
    { 12, -5 }        { :ok, { "fabien", "dagnat", 50 } }
    ```

  - else map

    ```
    %{ :red => 0xff0000, :green => 0x00ff00, :blue => 0x0000ff }
    %{ :first => "Fabien", :last => "Dagnat", :age => 50 }
    ```

- Unknown or variable quantity
  - access to head and tail ⇒ list

    ```
    [1, "a", :ok, false]
    ```

  - else map
- Raw data ⇒ binary for byte sequences

Strings

- Strings are binaries
  - efficient
  - concatenation by <> (like all binaries)
- litteral between "
- May contain several lines
- May contain interpolations
  `"3 + 0.14 = #{3 + 0.14}"` returns "3 + 0.14 = 3.14"

Pattern Matching

- Interest
  - all values are simple or composed fo simpler values
  - process a value often require to process its sub-values
  - often there is several case depending on the form of the value
- Principle
  - a set of match cases is defined
  - a match case is built from a pattern and an expression
  - one by one the match cases patterns are compared with a given value
  - if pattern match, its corresponding expression is executed
  - if no case match, an exception is raised
- A pattern is like a value but can contain variables
  - the comparison is made on the form of both the pattern and the value
  - if there is a match, the variables of the pattern are defined

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`								
`{:ok,1}`								
`{tutu,1}`								
`{:err,m}`								
`{log,i}`								
`{_,{f,a}}`								
`[]`								
`[hd|tl]`								
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`								
`{tutu,1}`								
`{:err,m}`								
`{log,i}`								
`{_,{f,a}}`								
`[]`								
`[hd|tl]`								
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`	✗	✗	✓	✗	✗	✗	✗	✗
`{tutu,1}`								
`{:err,m}`								
`{log,i}`								
`{_,{f,a}}`								
`[]`								
`[hd|tl]`								
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	1	0	{:ok,1}	{:err,"M"}	{"fd",{"F",50}}	[]	[1,:A,3]	[7,7]
1	✓	✗	✗	✗	✗	✗	✗	✗
{:ok,1}	✗	✗	✓	✗	✗	✗	✗	✗
{tutu,1}	✗	✗	tutu=:ok	✗	✗	✗	✗	✗
{:err,m}								
{log,i}								
{_,{f,a}}								
[]								
[hd|tl]								
[a,a|_]								
x								
_								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`	✗	✗	✓	✗	✗	✗	✗	✗
`{tutu,1}`	✗	✗	`tutu=:ok`	✗	✗	✗	✗	✗
`{:err,m}`	✗	✗	✗	`m="M"`	✗	✗	✗	✗
`{log,i}`								
`{_,{f,a}}`								
`[]`								
`[hd|tl]`								
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`	✗	✗	✓	✗	✗	✗	✗	✗
`{tutu,1}`	✗	✗	`tutu=:ok`	✗	✗	✗	✗	✗
`{:err,m}`	✗	✗	✗	`m="M"`	✗	✗	✗	✗
`{log,i}`	✗	✗	`log=:ok, i=1`	`log=:err, i="M"`	`log="fd", i={"F",50}`	✗	✗	✗
`{_,{f,a}}`								
`[]`								
`[hd|tl]`								
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`	✗	✗	✓	✗	✗	✗	✗	✗
`{tutu,1}`	✗	✗	`tutu=:ok`	✗	✗	✗	✗	✗
`{:err,m}`	✗	✗	✗	`m="M"`	✗	✗	✗	✗
`{log,i}`	✗	✗	`log=:ok, i=1`	`log=:err, i="M"`	`log="fd", i={"F",50}`	✗	✗	✗
`{_,{f,a}}`	✗	✗	✗	✗	`f="F" , a=50`	✗	✗	✗
`[]`								
`[hd|tl]`								
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	1	0	{:ok,1}	{:err,"M"}	{"fd",{"F",50}}	[]	[1,:A,3]	[7,7]
1	✓	✗	✗	✗	✗	✗	✗	✗
{:ok,1}	✗	✗	✓	✗	✗	✗	✗	✗
{tutu,1}	✗	✗	tutu=:ok	✗	✗	✗	✗	✗
{:err,m}	✗	✗	✗	m="M"	✗	✗	✗	✗
{log,i}	✗	✗	log=:ok, i=1	log=:err, i="M"	log="fd", i={"F",50}	✗	✗	✗
{_,{f,a}}	✗	✗	✗	✗	f="F" , a=50	✗	✗	✗
[]	✗	✗	✗	✗	✗	✓	✗	✗
[hd|tl]								
[a,a|_]								
x								
_								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`	✗	✗	✓	✗	✗	✗	✗	✗
`{tutu,1}`	✗	✗	`tutu=:ok`	✗	✗	✗	✗	✗
`{:err,m}`	✗	✗	✗	`m="M"`	✗	✗	✗	✗
`{log,i}`	✗	✗	`log=:ok, i=1`	`log=:err, i="M"`	`log="fd", i={"F",50}`	✗	✗	✗
`{_,{f,a}}`	✗	✗	✗	✗	`f="F" , a=50`	✗	✗	✗
`[]`	✗	✗	✗	✗	✗	✓	✗	✗
`[hd|tl]`	✗	✗	✗	✗	✗	✗	`hd=1` `tl=[:A,3]`	`hd=7` `tl=[7]`
`[a,a|_]`								
`x`								
`_`								

Exemples de filtrage de motif

pattern ↓ value →	1	0	{:ok,1}	{:err,"M"}	{"fd",{"F",50}}	[]	[1,:A,3]	[7,7]
1	✓	✗	✗	✗	✗	✗	✗	✗
{:ok,1}	✗	✗	✓	✗	✗	✗	✗	✗
{tutu,1}	✗	✗	tutu=:ok	✗	✗	✗	✗	✗
{:err,m}	✗	✗	✗	m="M"	✗	✗	✗	✗
{log,i}	✗	✗	log=:ok, i=1	log=:err, i="M"	log="fd", i={"F",50}	✗	✗	✗
{_,{f,a}}	✗	✗	✗	✗	f="F" , a=50	✗	✗	✗
[]	✗	✗	✗	✗	✗	✓	✗	✗
[hd|tl]	✗	✗	✗	✗	✗	✗	hd=1 tl=[:A,3]	hd=7 tl=[7]
[a,a|_]	✗	✗	✗	✗	✗	✗	✗	a=7
x								
_								

Exemples de filtrage de motif

pattern ↓ value →	`1`	`0`	`{:ok,1}`	`{:err,"M"}`	`{"fd",{"F",50}}`	`[]`	`[1,:A,3]`	`[7,7]`
`1`	✓	✗	✗	✗	✗	✗	✗	✗
`{:ok,1}`	✗	✗	✓	✗	✗	✗	✗	✗
`{tutu,1}`	✗	✗	`tutu=:ok`	✗	✗	✗	✗	✗
`{:err,m}`	✗	✗	✗	`m="M"`	✗	✗	✗	✗
`{log,i}`	✗	✗	`log=:ok, i=1`	`log=:err, i="M"`	`log="fd", i={"F",50}`	✗	✗	✗
`{_,{f,a}}`	✗	✗	✗	✗	`f="F" , a=50`	✗	✗	✗
`[]`	✗	✗	✗	✗	✗	✓	✗	✗
`[hd|tl]`	✗	✗	✗	✗	✗	✗	`hd=1` `tl=[:A,3]`	`hd=7` `tl=[7]`
`[a,a|_]`	✗	✗	✗	✗	✗	✗	✗	`a=7`
`x`	`x=1`	`x=0`	`x={:ok,1}`	`x={:err,"M"}`	`x={"fd",{"F",50}}`	`x=[]`	`x=[1,:A,3]`	`x=[7,7]`
`_`	✓	✓	✓	✓	✓	✓	✓	✓

Using pattern matching

- Three forms
  - p = e
  - **fn** $(p\ (,p)^*$ -> $e)^+$ **end** ou **fn** ($(p\ (,p)^*$) -> $e)^+$ **end**
    - all filter case must have the same arity
  - **case** e **do** $(p$ -> $e)^+$ **end**
- An example

```
case File.open("case.ex") do
 { :ok, file } ->
  IO.puts "First line: #{IO.read(file, :line)}"
 { :error, reason } ->
  IO.puts "Failed to open file: #{reason}"
end
```

Let's practice I (still on paper)

- A function receiving a couple and returning the couple swapping the elements of the received couple
- A function receiving two couples and returning a couple of whose first member is the couple of the two first elements and the second, the couple of the second elements
- A function receiving two functions `f` and `g` and a couple. It returns the couple made of the result of applying `f` to the first element and `g` to the second

Let's practice I (still on paper)

- A function receiving a couple and returning the couple swapping the elements of the received couple

  ```
  fn { fst, snd } -> { snd, fst } end
  ```

- A function receiving two couples and returning a couple of whose first member is the couple of the two first elements and the second, the couple of the second elements

- A function receiving two functions `f` and `g` and a couple. It returns the couple made of the result of applying `f` to the first element and `g` to the second

Let's practice I (still on paper)

- A function receiving a couple and returning the couple swapping the elements of the received couple

  ```
  fn { fst, snd } -> { snd, fst } end
  ```

- A function receiving two couples and returning a couple of whose first member is the couple of the two first elements and the second, the couple of the second elements

  ```
  fn { fst1, snd1 }, { fst2, snd2 } -> { { fst1,fst2 }, { snd1, snd2 } } end
  ```

- A function receiving two functions `f` and `g` and a couple. It returns the couple made of the result of applying `f` to the first element and `g` to the second

Let's practice I (still on paper)

- A function receiving a couple and returning the couple swapping the elements of the received couple

  ```
  fn { fst, snd } -> { snd, fst } end
  ```

- A function receiving two couples and returning a couple of whose first member is the couple of the two first elements and the second, the couple of the second elements

  ```
  fn { fst1, snd1 }, { fst2, snd2 } -> { { fst1,fst2 }, { snd1, snd2 } } end
  ```

- A function receiving two functions `f` and `g` and a couple. It returns the couple made of the result of applying `f` to the first element and `g` to the second

  ```
  fn f, g, { fst, snd } -> { f.(fst), g.(snd) } end
  ```

Let's practice II (still on paper)

- A function receiving two functions `f` and `g` and a couple. It return the result of applying `f` (resp. `g`) to the second member of the couple if the first is `:fst` (resp. `:snd`).

Let's practice II (still on paper)

- A function receiving two functions `f` and `g` and a couple. It return the result of applying `f` (resp. `g`) to the second member of the couple if the first is `:fst` (resp. `:snd`).

```
fn
  f, _g, { :fst, v } -> f.(v)
  _f, g, { :snd, v } -> g.(v)
end
```

Installation and documentation

https://elixir-lang.org/install.html

https://elixir-lang.org/docs.html

More practice

- Compute $a \Rightarrow b$ from a couple (a, b)

- A function indicating if a value is equal to zero

More practice

- Compute $a \Rightarrow b$ from a couple (a, b)

```
fn v ->
  case v do
    {true,true} -> true;     {true,false} -> false
    {false,true} -> true;    {false,false} -> true
  end
end
```

or

```
fn {true,x} -> x;    {false,_} -> true end
```

- A function indicating if a value is equal to zero

More practice

- Compute $a \Rightarrow b$ from a couple (a, b)

```
fn v ->
  case v do
    {true,true} -> true;     {true,false} -> false
    {false,true} -> true;    {false,false} -> true
  end
end
```

or

```
fn {true,x} -> x;    {false,_} -> true end
```

- A function indicating if a value is equal to zero

```
fn 0 -> true; _ -> false end
```

Closures

- A(n anonymous) function can use variables of its context of definition
- A closure = function + a set of variables (and their associated values)
- For example

```
iex(1)> const = 3
3
iex(2)> add_const = fn a -> a + const end
#Function<44.97283095/1 in :erl_eval.expr/5>
iex(3)> add_const.(2)
5
iex(4)> const = 8
8
iex(5)> add_const.(2)
5
```

Even more practice: a small problem

- Sometimes, it is useful to have a function table where atoms are associated with functions. Executing, in such a situation, consists in receiving an atom and arguments and returning the invocation of the function corresponding to the atom in the function table with the received argument.
  Define such a table and the associated `exec` function.

Even more practice: a small problem

- Sometimes, it is useful to have a function table where atoms are associated with functions. Executing, in such a situation, consists in receiving an atom and arguments and returning the invocation of the function corresponding to the atom in the function table with the received argument.
  Define such a table and the associated `exec` function.

```
function_table = %{ :add => fn a,b -> a+b end, :sub => fn a,b -> a-b end }
exec = fn k,{x,y} -> case function_table[k] do nil -> nil; f -> f.(x,y) end end
```

  then `exec.(:add,{1,2})` yields `3`, `exec.(:sub,{1,2})` yields `-1` and `exec.(:sub,{1,2})` yields `nil`

Named functions and modules

- Functions can be named (using *snake_case* like variables)
- Named function must defined within a module (name in *CamelCase*)

```
defmodule Complex do
  def add({ r1, i1 }, { r2, i2 }) do { r1+r2, i1+i2 } end
  def mul({ r1, i1 }, { r2, i2 }) do
    { r1*r2 - i1*i2, r1*i2 + i1*r2 }
  end
end
```

- Arguments
  - optional parentheses
  - no space before the opening parentheses!
- Call with `Complex.add({ 1, 0 },{ 0, 1 })`
  - optional parentheses

More on named functions I

- Several matching cases

```
defmodule ErrorLogger do
  def log({:ok,_}) do ... end
  def log({:error, msg}) do IO.puts("Error: " <> msg) end
end
```

- Arity may vary (arity is part of the function identifier)

```
defmodule Activity do
  def wait do ... end
  def wait(timeout) do ... end
end
```

More on named functions II

- Optional parameters with default values (generate a function with several arities)

```
defmodule Activity do
  def wait(timeout \\ 1000) do ... end
end
```

- Private functions are defined using **defp**

Difference between anonymous and named functions

- Detailed explanation: `https://stackoverflow.com/a/18023790`
- Anonymous
  - capture variables (closures)
  - fixed arity
  - values like others
  - call with a dot
- Named
  - not closures
  - multiple arity and optional parameters
  - not values
    - contained in modules (definition and use)

    $\Rightarrow$ operator `&` : `&hello/1`
  - call without dot

Repeating things

- No loop
  - recursive functions (therefore named)

```
defmodule MyList do
  def len [] do 0 end
  def len [_|tl] do 1 + len(tl) end
end
```

  - by using a predefine iteration function

```
Enum.reduce([1, 2, 3], 0, fn _, acc -> 1 + acc end)
```

Repeating things

- Concept of terminal recursion

```
defmodule MyListTerm do
  def len l do len(0,l) end
  defp len n,[] do n end
  defp len n,[_|tl] do len(n+1,tl) end
end
```

Recursion practice

- Propose a function to compute the Fibonacci numbers

- Propose a function `filter` that receives a list `l` and a function `f` and returns the list of elements of `l` that returns true when applied to `f`.

Recursion practice

- Propose a function to compute the Fibonacci numbers

```
def fibonacci 0 do 0 end
def fibonacci 1 do 1 end
def fibonacci n do fibonacci(n-1) + fibonacci(n-2) end
```

- Propose a function `filter` that receives a list `l` and a function `f` and returns the list of elements of `l` that returns true when applied to `f`.

Recursion practice

- Propose a function to compute the Fibonacci numbers

```
def fibonacci 0 do 0 end
def fibonacci 1 do 1 end
def fibonacci n do fibonacci(n-1) + fibonacci(n-2) end
```

- Propose a function `filter` that receives a list `l` and a function `f` and returns the list of elements of `l` that returns true when applied to `f`.

```
def filter [], _f do [] end
def filter [head|tail], f do
 case f.(head) do
  true -> [head| filter(tail, f)]
  _ -> filter(tail, f)
 end
end
```

A larger exercise

A binary tree is either empty or contain a value and have exactly two children, which are themselves binary trees (one is called the left child and the other the right child).
A binary search tree also has the following invariant: for any node, all values contained in the left subtree are less than the value of the node, and all values in the right subtree are greater than that value.

1. Provide an `add` function that adds an integer value to a binary search tree.
2. Use this function to sort a list of integers.

Conclusion

- We started to explore the functional core of Elixir
  - start reading sections 1 and 2 of the site
  - finish all exercices of the sildes
- Next week, we start using Elixir seriously in Fiab
  - More on Elixir and more practice
  - Learning by/through practice