# what is functional programming?

- Functional programming is a programming paradigm built upon the notion of mathematical function. Every program is considered a function from input to output.
- One of the most outstanding features of FP is that it is side-effect free. Definitions are immutable, which allows for equational reasoning. This is opposed to imperative programming, which is based on assignment.
- Another difference with imperative programming is that functional data structures are mostly tree-like, as opposed to the random access structures (arrays) found in most imperative languages.
- Functional programming languages are higher-order. Functions are *first-class* citizens: they can act as inputs to other functions, be returned as result of functions and be part of data structures.
- Most functional programming languages posess rich type systems that allow the programmer to understand what programs do, help compilation and help detect errors at compile time.
- Syntax tries to be close to mathematical notation, and intensional definitions over (possibly infinite) data domains are allowed.

POLITÉCNICA

# (de)motivation

## how functional is Elixir?

- Built upon the notion of mathematical function. YES
- Side-effect free YES
- Tree-like data structures YES
- Higher order YES
- Type system NO!
- Clean, concise notation HMMM
- Intensional definitions YES
- Infinite data structures NO

POLITÉCNICA

# bits of history

- 1930s: Alonzo Church develops the lambda calculus and type systems.
- 1950s: John McCarthy introduces the LISP programming language. Based on lambda calculus but allows for destructive assignment.
- 1960s: John Landin introduces ISWIM, first pure functional language.
- 1970s: John Backus proposes FP, a functional programming language oriented towards the use of higher-order operators and algebraic program transformation.
- 1970s: Robin Milner introduces ML and the Milner type system.
- 1970s – 1980s: David Turner concludes the definition of Miranda a (lazy) functional programming language that tried to be the top-of-the-top fp language...
- 1987: A group of rebels introduce Haskell, an open alternative to Miranda later to become the standard pure fp language.
- 1990s, 2000s: Increasing influence on mainstream programming languages.

POLITÉCNICA

# plan

- Today I will introduce the functional programming that would be more or less covered in any fp course, and next week I will focus on more Elixir-specific features.

POLITÉCNICA

## defining functions

- Named functions:

```
def square(n) do
  n * n
end
```

- Anonymous function definitions, first syntax:

```
iex(1)> square = fn(x) -> x * x end
#Function<44.79398840/1 in :erl_eval.expr/5>
iex(2)> square.(2)
4
```

- Anonymous functions, alternative syntax (indexical)

```
iex(1)> square = &(&1 * &1)
#Function<44.79398840/1 in :erl_eval.expr/5>
iex(2)> square.(3)
9
```

# recursive data structures

- list processing, pattern matching, induction
- ```
  def is_sorted([]) do
      true
  end

  def is_sorted([_]) do
      true
  end

  def is_sorted([x,y|zs]) do
      x <= y && is_sorted([y|zs])
  end
  ```
- Insertion trees

  ```
  {:node, {:node, :tip, 1, :tip}, 2, {:node, :tip, 3, :tip}}
  ```

POLITÉCNICA
  ```

# higher-order: functions as input to other functions

- Imagine that need some *ad-hoc* way of comparing elements – not just the standard ordering on Elixir objects.
- We can modify our `is_sorted/1` function so that it receives a binary predicate that will be used to compare consecutive elements:
- ```
  def is_sorted2(p,[x,y|zs]) do
      p.(x,y) && is_sorted2(p,[y|zs])
  end

  def is_sorted2(_,xs) do
      is_sorted(xs)
  end
  ```
- ```
  iex(3)> Scratch.is_sorted2(&(&2 < &1), [3,2,1])
  true
  iex(4)> Scratch.is_sorted2(&(&2 < &1), [1,2,3])
  false
  ```

POLITÉCNICA

# higher-order: functions as output from other functions

- Composition:

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

- ```
  iex(2)> square = fn(x) -> x * x end
  #Function<44.79398840/1 in :erl_eval.expr/5>
  iex(3)> fourth = Scratch.comp(square,square)
  #Function<0.130864331/1 in Scratch.comp/2>
  iex(4)> fourth.(2)
  16
  ```

- Partial application:

- ```
  iex(1)> twice = &(2 * &1)
  #Function<44.79398840/1 in :erl_eval.expr/5>
  iex(2)> twice.(3)
  6
  ```

POLITÉCNICA

# higher-order list operators: fold

- Summation:

```
def sum([]) do
    0
end
def sum([x|xs]) do
    x + sum(xs)
end
```

- Concatenation:

```
def append([],ys) do
    ys
end
def append([x|xs],ys) do
    [x|append(xs,ys)]
end
```

- Common pattern: foldr

$$foldr([x_1, x_2, x_3], b, op) = op(x_1, op(x_2, op(x_3, b)))$$

- Exercise: rewrite `sum` and `append` using `List.foldr`

POLITÉCNICA

# higher-order list operators: map

- A *map* pattern transforms a list by applying the same function to all its elements:

$$map([x_1, x_2, x_3], f) = [f(x_1), f(x_2), f(x_3)]$$

- A *map* pattern is a special case of a *foldr*. Exercise: define your own *map* operator using `List.foldr`.
- Example:

```
iex(3)> Enum.map([1,-2,3,-42],&(abs(&1)))
[1, 2, 3, 42]
```

- Exercise: Compute the cartesian product of two lists:

$$cart([x_1, x_2, x_3], [y_1, y_2]) = [(x_1, y_1), (x_1, y_2), \ldots, (x_3, y_2)]$$

- Maps can be defined over other `Enumerable` types:
  - maps over `Range`s:

```
iex(3)> Enum.map(1..10,fn x -> x*x end)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
iex(4)> cart(1..2,1..3)
[{1, 1}, {1, 2}, {1, 3}, {2, 1}, {2, 2}, {2, 3}]
```

  - maps over `Map`s:

```
iex(2)> me = %{height: 165, weight: 69}
%{height: 165, weight: 69}
iex(3)> Enum.map(me,fn {k, v} -> if k == :weight do {:weight, v - 5}
    else {k, v} end end)
[height: 165, weight: 64]
```

The Elixir for losing weight!

POLITÉCNICA

# higher-order list operators: filter

- Filter removes those elements for which a given property does not hold.
- 
```
iex(8)> Enum.filter(1..100,&(rem(&1,2)==0))
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62,
 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92,
 94, 96, 98, 100]
```
- Filters are also a special case of a fold. Exercise: Define your own version of `filter` using `List.foldr`.

POLITÉCNICA

# higher-order list operations: zipping

- Zipping is combining two lists – of the same length – elementwise:

$$zip([x_1, x_2, x_3], [y_1, y_2, y_3]) = [(x_1, y_1), (x_2, y_2), (x_3, y_3)]$$

- Zipping in Elixir is provided by the function `Enum.zip`:

```
iex(1)> Enum.zip([1,2],[3,4])
[{1, 3}, {2, 4}]
```

- Most of the times we are interested in zipping two lists with some function:

$$zipWith([x_1, x_2, x_3], [y_1, y_2, y_3], f) = [f(x_1, y_1), f(x_2, y_2), f(x_3, y_3)]$$

  This allows, for example, to add two n-dimensional vectors, etc.

- Exercise: Implement your own `zipWith/3` using `Enum.zip/2` and other higher-order list operations.

POLITÉCNICA

# laws and algebraic reasoning

- The higher-order list operators that we have introduced satisfy many remarkable equalities. For example, maps are *list homomorphisms*:

  $$map(f) \circ map(g) = map(f \circ g)$$

  In Elixir's syntax:

  ```
  List.map(List.map(xs,g),f) == List.map(xs,comp(f,g))
  ```

- Laws offer us a way of transforming programs into possibly more efficient versions.
- Other well known laws:
  - $filter(p) \circ filter(q) = filter(p \wedge q)$
  - $fold(map(xs, f), b, op) = fold(xs, b, \lambda x, y.op(f(x), y))$

POLITÉCNICA

# higher-order list operations: left folding

- Right folding (foldr) is not the only possible way of combining all the elements in a sequence.
- For example, `List.foldl/3` operates some initial value with the head of a list and uses the result obtained to reduce the rest:

$$List.foldl([x_1, x_2, x_3], b, f) = f(x_3, f(x_2, f(x_1, b)))$$

- One advantage of left folding is that it can be naturally implemented using *tail recursion*:

```
def foldl([h|ts],b,f) do
  foldl(ts,f(h,b),f)
end
def foldl([],b,_) do
  b
end
```

- We can see that the following law holds:

$$foldl(xs, b, f) = foldr(reverse(xs), b, f)$$

- Corollary: When *f* is commutative and associative *foldr* and *foldl* are interchangeable.
- Exercise: Use the law above to derive a concise and efficient definition for list reversal.

POLITÉCNICA

# comprehensions

- In mathematics we often find definitions like this one:

$$\{(x, y, z) \cdot x, y, z \in 1 \ldots 100 \mid x^2 + y^2 = z^2\}$$

- Exercise: Implement the expression above using higher-order list operations.
- Elixir allows for a more concise notation for this kind of definitions:

```
for x <- 1..100, y <- 1..100, z <- 1..100, x*x + y*y == z*z,
    do: {x,y,z}
```

POLITÉCNICA