Functional Programming with Elixir

Adolfo Neto

June 2025

Presentation Objectives

- Introduce the main concepts of functional programming
- Show how these concepts are applied in Elixir
- Present practical code examples
- Compare with other paradigms
- Discuss benefits and challenges

What is Functional Programming?

- Paradigm based on the use of pure functions
- Avoids mutable state and side effects
- Emphasis on:
  - Immutability
  - Recursion
  - Higher-order functions
  - Composition

What are functions?

- Carnielli-Epstein

Elixir: a functional language

- Created by José Valim
- Runs on the BEAM (Erlang virtual machine)
- Robust support for concurrency and fault tolerance
- Modern syntax, inspired by Ruby
- Functional from the core

Named Functions

```elixir
defmodule Mathematics do
  def double(x), do: x * 2
  def square(x), do: x * x
end

IO.puts Mathematics.double(5)      # 10
IO.puts Mathematics.square(4)      # 16
```

- Organized in modules
- Called with module name and dot
- Can have multiple clauses

Functions with Pattern Matching (and multiple clauses)

```elixir
defmodule Factorial do
  def calculate(0), do: 1
  def calculate(n), do: n * calculate(n - 1)
end

IO.puts Factorial.calculate(5)  # 120
```

- Direct pattern matching in parameters
- Powerful alternative to `if`/`else`

Anonymous Functions

```elixir
double = fn x -> x * 2 end
IO.puts double.(10)  # 20
```

- Created with `fn -> end`
- Executed with `.()`
- Useful for higher-order functions

Higher-Order Functions

```elixir
apply = fn f, value -> f.(value) end

increment = fn x -> x + 1 end

IO.puts apply.(increment, 9)  # 10
```

- Receive or return functions
- Foundation of `Enum` and `Stream`

Function Composition

```elixir
multiply_by_2 = fn x -> x * 2 end
add_3 = fn x -> x + 3 end

compose = fn f, g ->
  fn x -> f.(g.(x)) end
end

new_function = compose.(multiply_by_2, add_3)

IO.puts new_function.(4)  # (4 + 3) * 2 = 14
```

- Composition: f(g(x))
- Creates more complex functions from simple ones

Variables are Immutable

```
x = 10
x = x + 1  # This does not reassign: creates a new `x`

IO.puts x  # 11
```

- No state mutation
- Reduces bugs in concurrent environments

Lists and Enum

```elixir
list = [1, 2, 3, 4]

Enum.map(list, fn x -> x * 2 end)
|> IO.inspect  # [2, 4, 6, 8]
```

- Functional operations on collections
- `Enum`, `Stream`, `MapSet`, etc.

Pipelines (|>)

```
" elixir "
|> String.trim()
|> String.upcase()
|> IO.puts()  # ELIXIR
```

- Makes code more readable
- Passes the result of one function as the argument to the next

Recursion Instead of Loops

```elixir
defmodule Counter do
  def count(0), do: IO.puts("End")
  def count(n) do
    IO.puts(n)
    count(n - 1)
  end
end

Counter.count(3)
```

- Natural feature in functional languages
- Tail-call optimization

Match and Guards

```elixir
defmodule Parity do
  def check(n) when rem(n, 2) == 0, do: "even"
  def check(_), do: "odd"
end

IO.puts Parity.check(4)  # even
IO.puts Parity.check(3)  # odd
```

- `when` restricts clause execution
- Helps avoid complicated `if` logic

Closures

```
multiplier_factory = fn factor ->
  fn x -> x * factor end
end

multiply_by_10 = multiplier_factory.(10)
IO.puts multiply_by_10.(5)  # 50
```

- Functions that capture the environment where they were defined
- Basis for currying and generating custom functions

Comparison with OO

OO Concept	Functional Equivalent
Object	Structure + pure functions
Methods	Functions in modules
Mutable state	Immutable data
Inheritance	Function/module composition
Loops	Recursion

Advantages of Functional Programming

- Fewer side effects
- Safe concurrency
- More declarative and readable code
- Easier testing

Disadvantages / Challenges

- Initial learning curve
- Different mindset from imperative
- Performance in some contexts may be lower
- Debugging long pipelines can be difficult

Elixir in Practice

- Used by companies like Discord and Remote
- Excellent for distributed and fault-tolerant applications
- Productivity with `mix`, `iex`, `Phoenix`, `LiveView`

Conclusion

- Elixir is a great entry point to functional programming
- Leverages BEAM's robustness with friendly syntax
- Encourages good practices from the start
- Start by writing pure functions, using `Enum`, `Stream`, `|>`, and exploring `iex`

References

- Elixir School
- Programming Elixir 1.6 – Dave Thomas
- Learn You Some Erlang
- Official documentation: https://elixir-lang.org

How to generate a PDF:

1. Save this content as `slides.md`
2. Generate the PDF:

```
pandoc -t beamer -V theme:metropolis -o slides.pdf slides.m
```

Some links

[1] https://www.youtube.com/watch?v=b8q3CRsfi2M

[2] https://www.rocketseat.com.br/blog/artigos/post/programacao-funcional-guia-completo

[3] https://www.alura.com.br/curso-online-elixir-sintaxe-programacao-funcional-pattern-matching

[4] https://www.youtube.com/watch?v=53Lv3efp7Rk

[5] https://www.youtube.com/watch?v=wJoo7Yicu5g

[6] https://pt.wikipedia.org/wiki/Elixir_(linguagem_de_programa%C3%A7%C

[7] https://www.eca.usp.br/acervo/producao-academica/002835871.pdf

[8] https://repositorio.mcti.gov.br/bitstream/mctic/5184/1/2010_quimica_ve 2030.pdf