Programação Funcional na Web com Elixir e Elm


rhnonose


RodrigoNonose

DIGITAL
NATIVES

Orientação a Objeto

- Objetos contém estado
- Um objeto pode mandar mensagens
- Um objeto pode receber mensagens

Problemas OO

- Estado + Lógica no mesmo lugar
- Mutabilidade de estado em qualquer lugar do método dentro do objeto
- Mutabilidade de estado fora do objeto
- Acoplamento temporal

Problemas OO

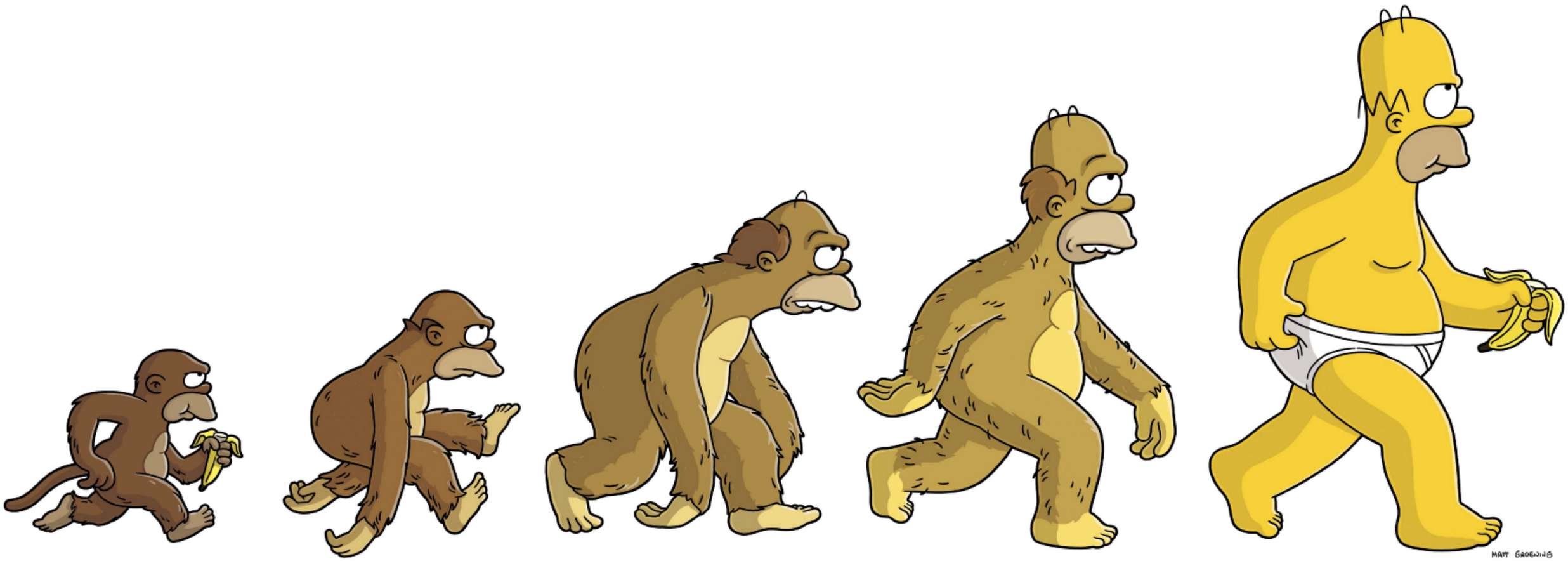
- Herança não cumpre a promessa de reusabilidade

- Passagem de objetos por referência quebra encapsulamento

Padrões OOP

- Singleton
- Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Adapter
- Decorator
- Bridge
- Proxy

Programação Funcional



Elm

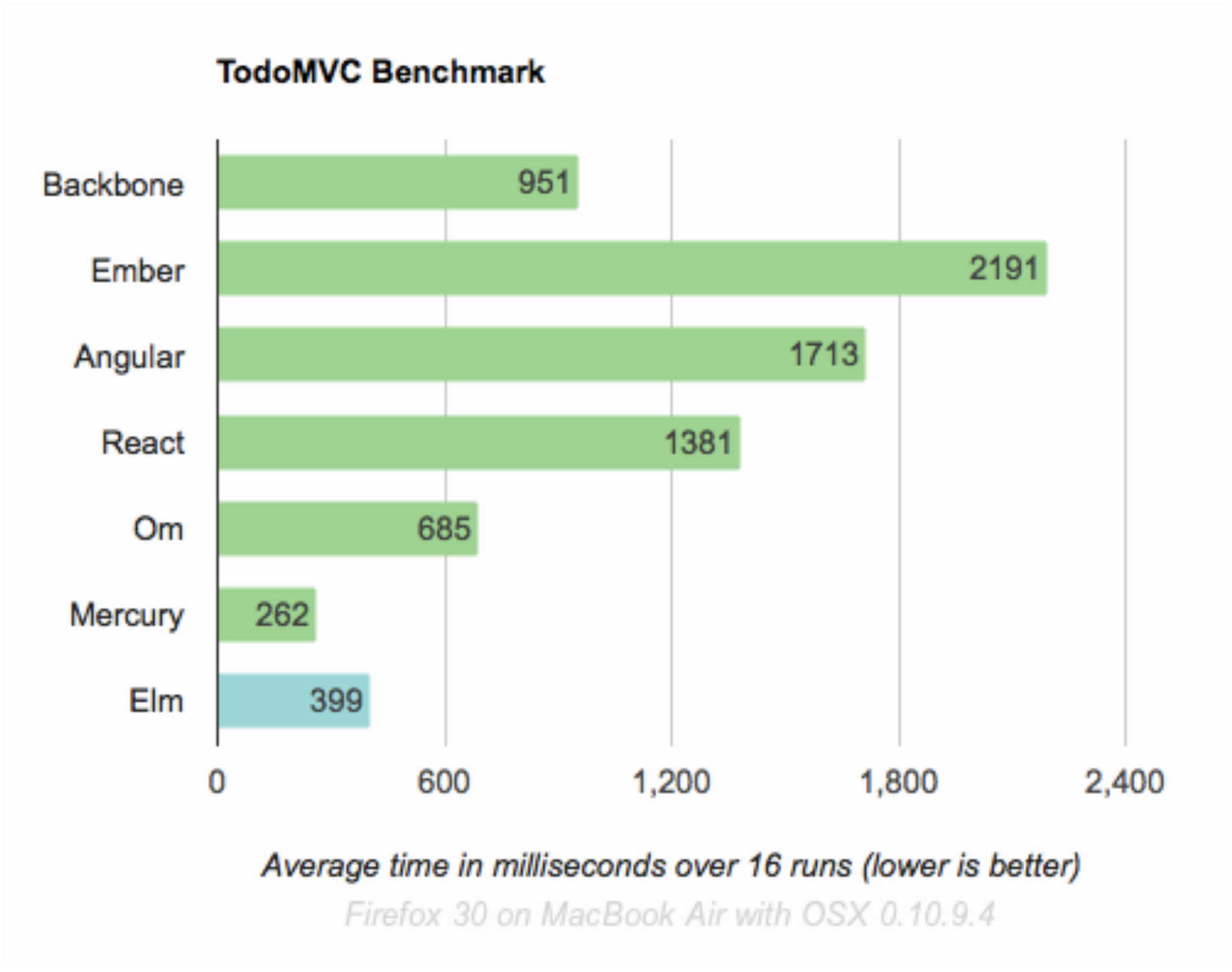
Elm

- Puramente funcional, estaticamente e fortemente tipada
- Expressiva, concisa e autodocumentada
- Imutabilidade e Transparência Referencial

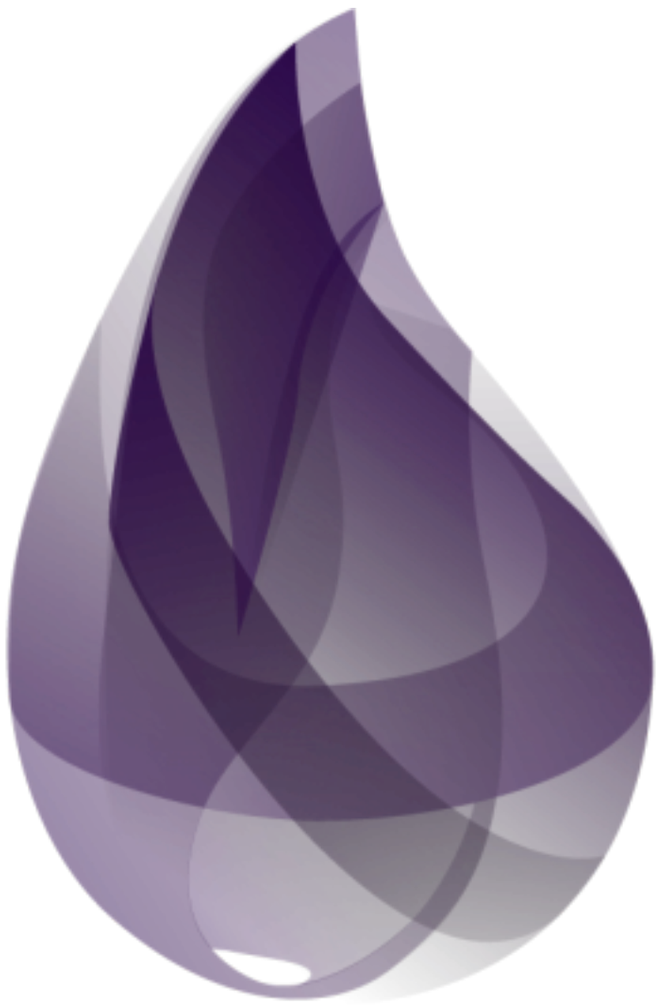
Elm

- Não existe *null*
- Não existe *undefined function*
- Interoperável com Javascript
- Versionamento Semântico Forçado

Elm


TodoMVC Benchmark
Backbone
951
Ember
2191
Angular
1713
React
1381
Om
685
Mercury
262
Elm
399
0
600
1,200
1,800
2,400
Average time in milliseconds over 16 runs (lower is better)
Firefox 30 on MacBook Air with OSX 0.10.9.4

Elixir



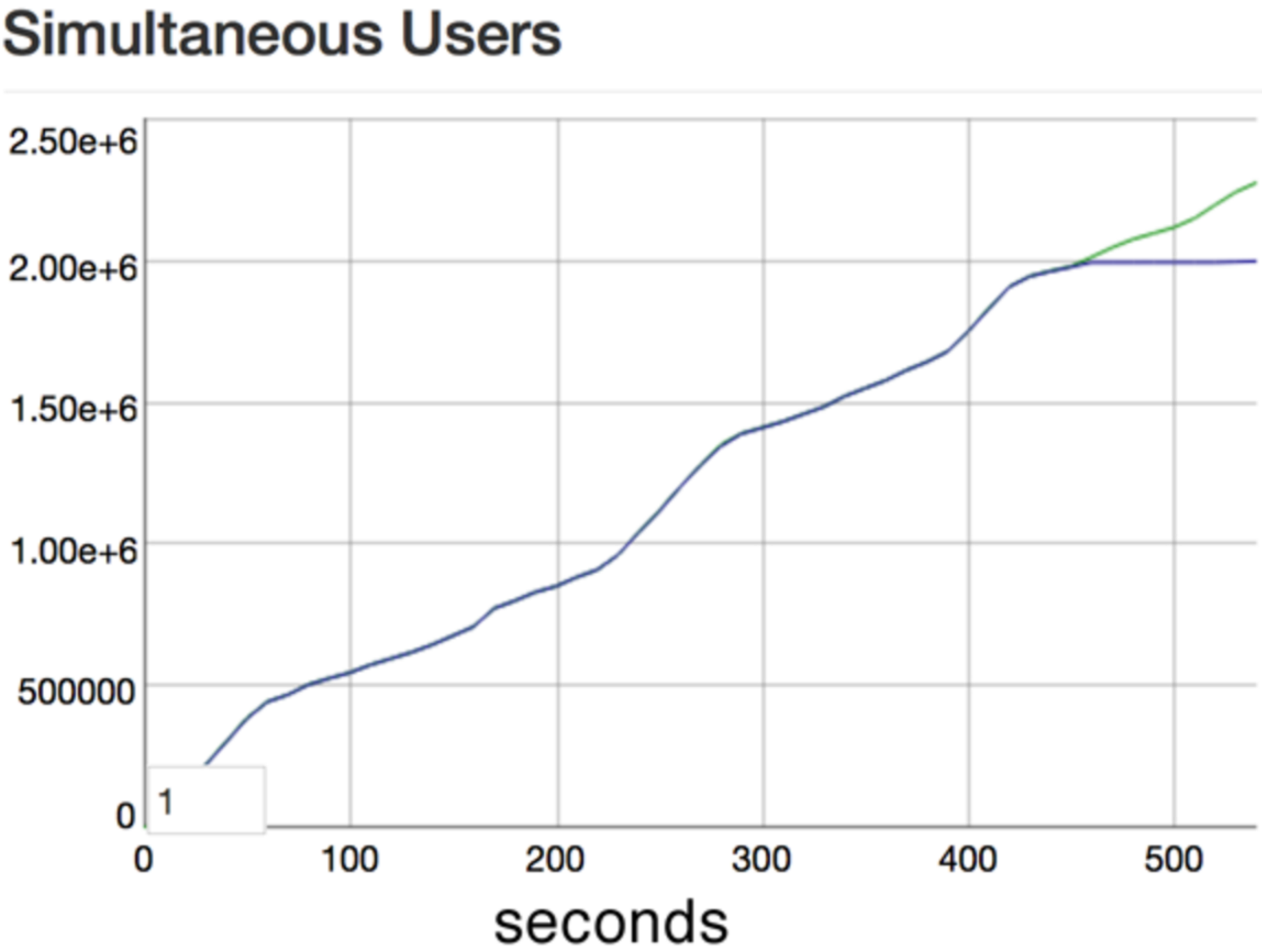
Elixir

- Funcional, dinamicamente tipada
- Foco em escalabilidade e manutenibilidade
- Tolerante a falhas
- Extensível
- Compilada para a *Erlang Virtual Machine* (*BEAM*)

Elixir

Framework	Throughput (req/s)	Latency (ms)	Consistency (σ ms)
Plug	198328.21	0.63ms	2.22ms
Phoenix 0.13.1	179685.94	0.61ms	1.04ms
Gin	176156.41	0.65ms	0.57ms
Play	171236.03	1.89ms	14.17ms
Phoenix 0.9.0-dev	169030.24	0.59ms	0.30ms
Express Cluster	92064.94	1.24ms	1.07ms
Martini	32077.24	3.35ms	2.52ms
Sinatra	30561.95	3.50ms	2.53ms
Rails	11903.48	8.50ms	4.07ms

Elixir


Simultaneous Users
2.50e+6
2.00e+6
1.50e+6
1.00e+6
500000
0
1
0
100
200
300
400
500
seconds

Programação Funcional

- Funções Puras
- Recursão
- Funções de Primeira Classe
- Funções de Alta Ordem

Funções Puras

Javascript

```
1   // com efeito colateral
2   contador = 0
3   function incremento(){
4       contador++;
5   }
6   incremento();
7   console.log(contador); // 1
8
9   // sem efeito colateral
10  function incremento_puro(contador){
11      return contador + 1;
12  }
13  novo_contador = incremento_puro(contador);
14  console.log(novo_contador) // 2
15  console.log(contador) // 1
```

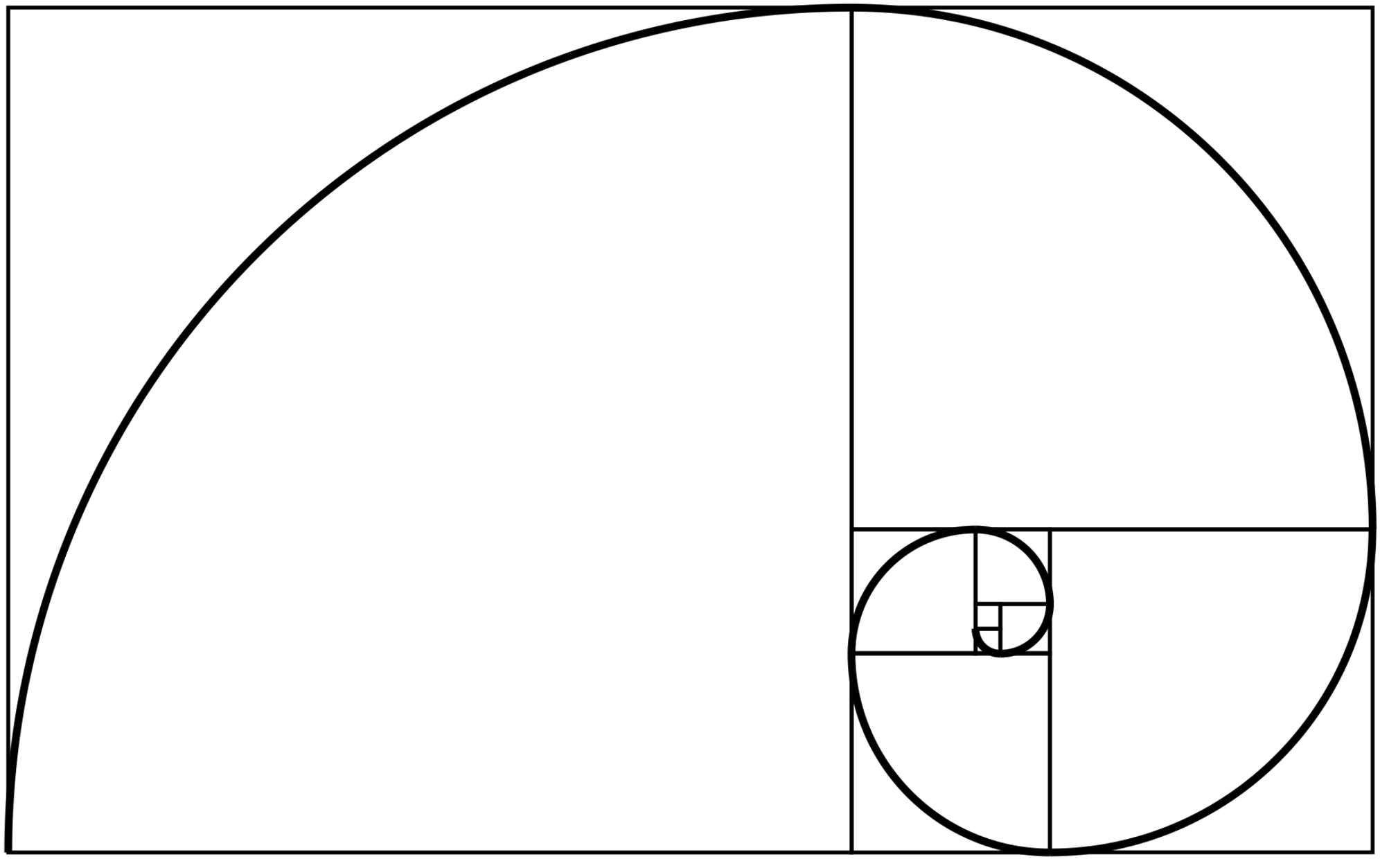
Elm

```
> a = 0
0 : number
> a = a + 1
-- BAD RECURSION ---------------------------------------- repl-temp-000.elm

`a` is defined directly in terms of itself, causing an infinite loop.

2| a = a + 1
   ^
Maybe you are trying to mutate a variable? Elm does not have mutation, so when I
see `a` defined in terms of `a`, I treat it as a recursive definition. Try
giving the new value a new name!

Maybe you DO want a recursive value? To define `a` we need to know what `a` is,
so let's expand it. Wait, but now we need to know what `a` is, so let's expand
it... This will keep going infinitely!

To really learn what is going on and how to fix it, check out:
<https://github.com/elm-lang/elm-compiler/blob/0.18.0/hints/bad-recursion.md>
```

Elixir

```elixir
a = 0
#declaração
incremento = fn ->
  a = a + 1
end
incremento.() # 1
a # 0

```

Recursão



Java

```java
//fibonacci iterativo
static int fibo_iter(int n) {
    int x = 0, y = 1, z = 1;
    for (int i = 0; i < n; i++) {
        x = y;
        y = z;
        z = x + y;
    }
    return x;
}

//fibonacci recursivo
static int fibo_recur(int n) {
    if ((n == 1) || (n == 0)) {
        return n;
    }
    return fibo_recur(n - 1) + fibo_recur(n - 2);
}
```

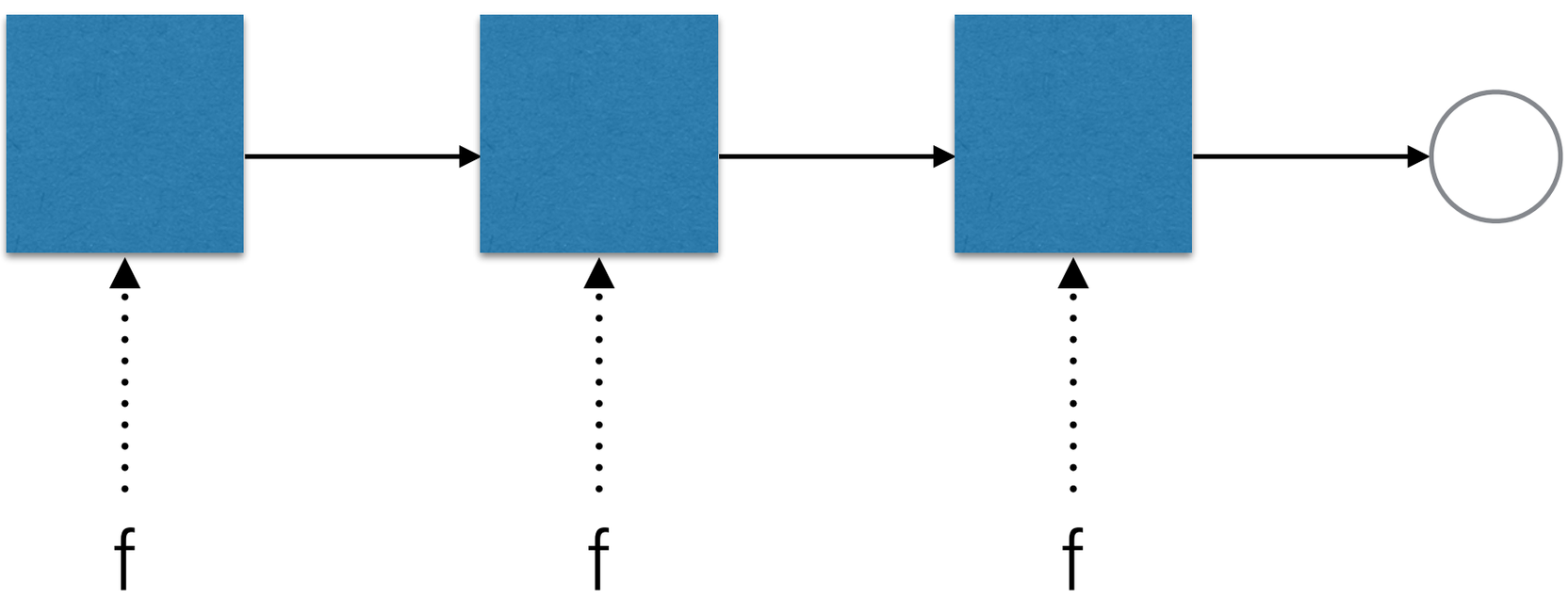
Elm

```
fibo n =
  case n of
    0 -> 0
    1 -> 1
    n -> (fibo (n - 1)) + (fibo (n - 2))
```

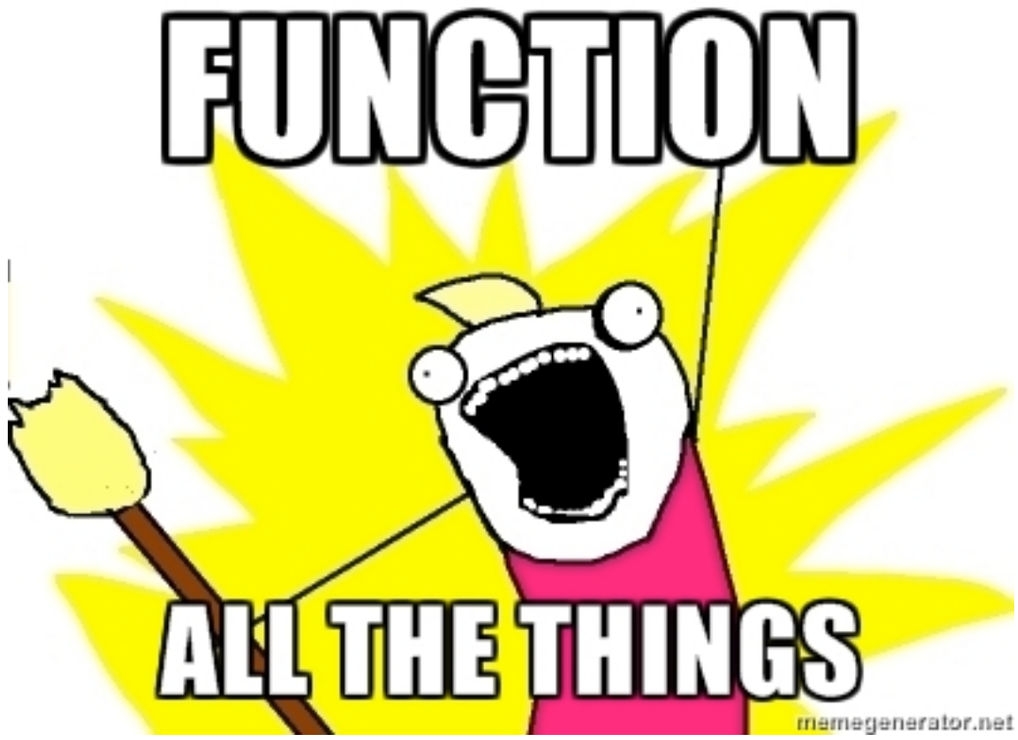
Elixir

```elixir
def fibo(0), do: 0
def fibo(1), do: 1
def fibo(n) do
  fibo(n - 1) + fibo(n - 2)
end
```

Lista Ligada



Funções de Primeira Classe



Elm

```elm
-- declara função superior
aplicar a b fun =
  fun a b
-- declara a função a ser passada por parâmetro
soma a b = a + b

-- chama a função
aplicar 1 2 (soma) -- 3

-- chama a função com outros parâmetros
aplicar "nome" "sobrenome" (++) -- "nomesobrenome"
```

Elixir

```elixir
# declara função superior
aplicar = fn a, b, fun -> fun.(a, b) end
# declara a função a ser passada por parâmetro
soma = fn a, b -> a + b end

# chama a função
aplicar.(1, 2, soma) # 3

# chama a função com outros parâmetros
aplicar.("nome", "sobrenome", &<>/2) #"nomesobrenome"
```

Funções de Alta Ordem

Java

```java
//"Função" para filtrar valores ímpares
public List<Integer> filtraImpares(List<Integer> lista) {
    List<Integer> listaFiltrada = new ArrayList<Integer>();

    for(Integer elemento : lista) {
        if(1 == elemento % 2) {
            listaFiltrada.add(elemento);
        }
    }
    return listaFiltrada;
}
```

```java
//"Função" com lógica parametrizada
public List<Integer> filtra(List<Integer> lista, Predicado predicado) {
    List<Integer> listaFiltrada = new ArrayList<Integer>();

        for (Integer elemento : lista) {
            if (predicado.avaliar(elemento)) {
                listaFiltrada.add(elemento);
            }
        }
        return listaFiltrada;
}

public interface Predicado {
    public boolean avaliar(Integer argumento);
}

class EhPar implements Predicado {

    public boolean avaliar(Integer argumento) {
        return 0 == argumento % 2;
    }
}

class EhImpar implements Predicado {

    public boolean avaliar(Integer argumento) {
        return 1 == argumento % 2;
    }
}
```

Elixir

```elixir
#função anônima 1
Enum.filter([1,2,3,4], fn n -> rem(n, 2) == 0 end)
Enum.filter([1,2,3,4], fn n -> rem(n, 2) == 1 end)

#função anônima 2
Enum.filter([1,2,3,4], &(rem(&1, 2) == 0))
Enum.filter([1,2,3,4], &(rem(&1, 2) == 1))

#função existente do módulo Integer
Enum.filter([1,2,3,4], &Integer.is_even/1)
Enum.filter([1,2,3,4], &Integer.is_odd/1)
```

Funções de Alta Ordem

- Map
- Reduce
- Filter
- Find
- Split

Java 8

```java
//Java 8
Arrays
    .asList("elm", "elixir", "scala", "clojure", "haskell")
    .stream()
    .filter(str -> !"elixir".equals(str))
    .collect(Collectors.toList());
```

Aplicação parcial

```haskell
-- função com 3 parâmetros de entrada
add a b c = a + b c

-- chamar sem todos os parâmetros
add1 = add 1 -- <function> : number -> number -> number

-- chamar a função retornada
add2 = add1 1 -- <function> : number -> number

-- chamar pela "terceira" vez
add2 1 -- 3
```

Piping

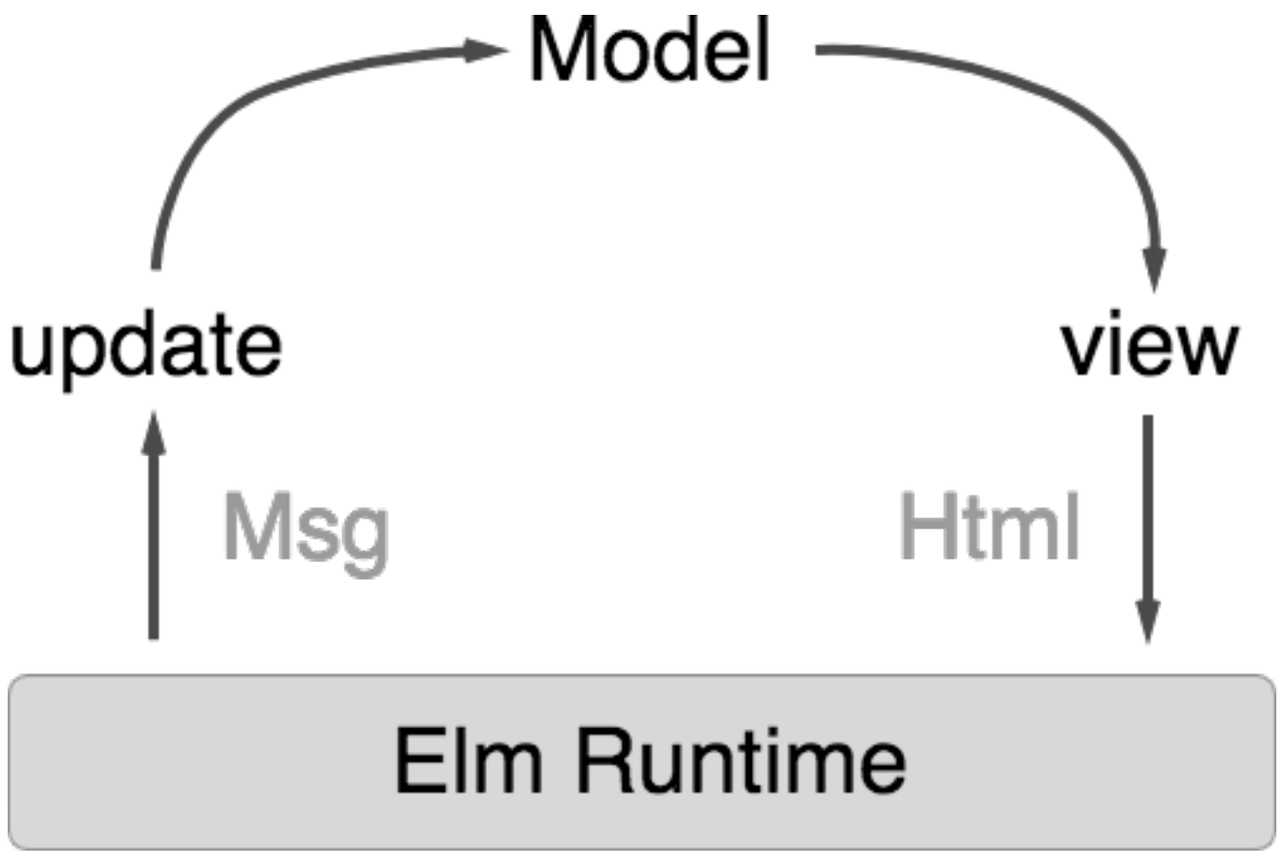
```elixir
Enum.map(
  String.split(
    String.downcase(
      "ELIXIR-ELM"),
    "_"),
  &String.capitalize/1)
# ["Elixir", "Elm"]
```

Piping

```elixir
Enum.map(
  String.split(
    String.downcase(
      "ELIXIR-ELM"),
    "_"),
  &String.capitalize/1)
# ["Elixir", "Elm"]

"ELIXIR-ELM"
|> String.downcase()
|> String.split("_")
|> Enum.map(&String.capitalize/1)
# ["Elixir", "Elm"]
```

Pattern Matching

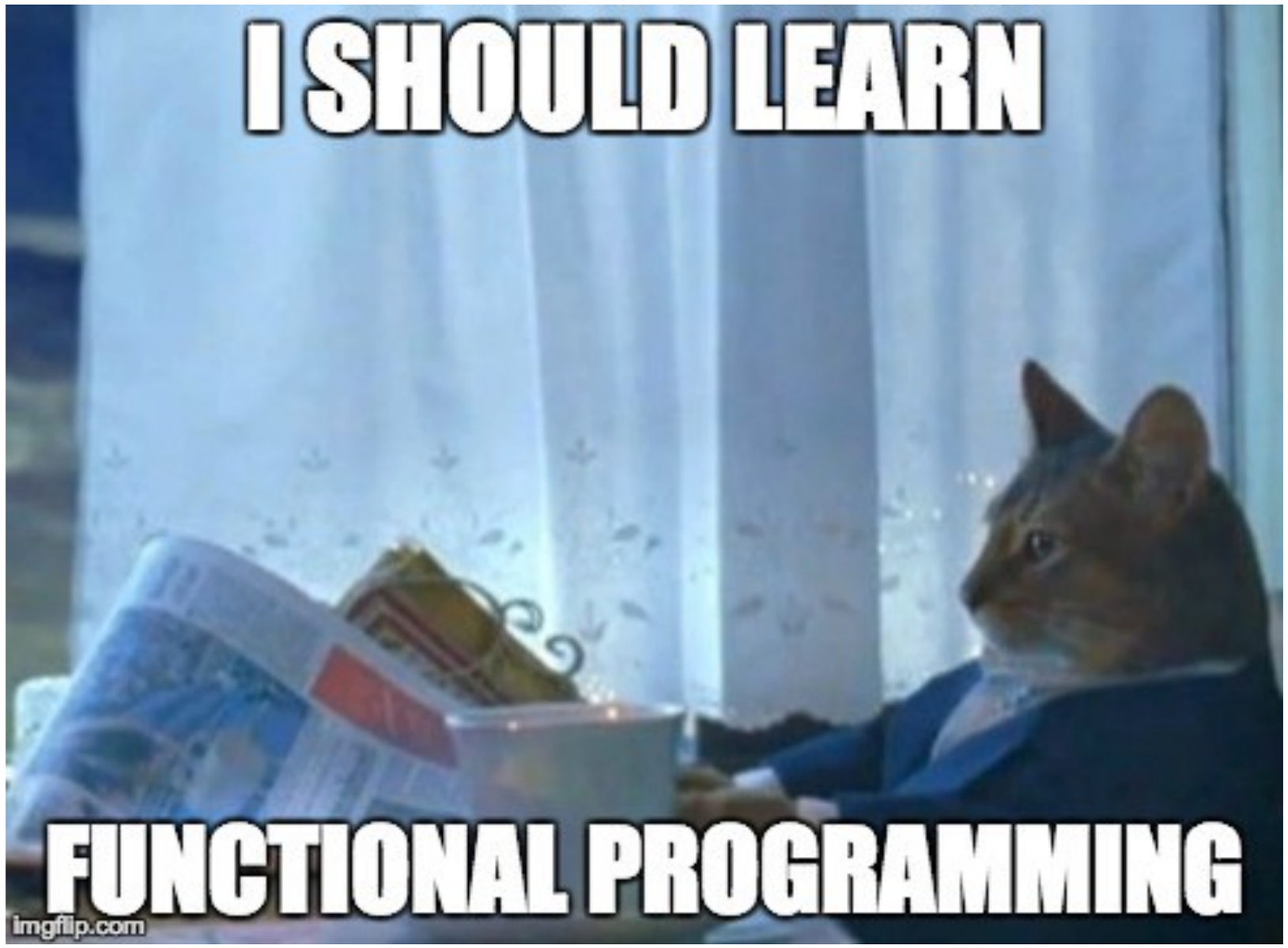
Arquitetura Elm



POO vs PF

- Mais abstrações
- Requer mais disciplina e treinamento
- Estado escondido e mutável
- Dependente de estado

- Menos abstrações
- Mais fácil de programar sem ter muita experiência
- Estado exposto e imutável
- Independente de estado


I SHOULD LEARN
FUNCTIONAL PROGRAMMING
imgflip.com

Obrigado!