



**CENTRO UNIVERSITÁRIO CARIOCA**

**CLAUDIO FÉLIX**

**MOACIR BONIFACIO AFFONSO**

**LINGUAGEM ELIXIR PARA APRENDIZAGEM DE PROGRAMAÇÃO  
CONCORRENTE**

Rio de Janeiro

2021

CLAUDIO FÉLIX  
MOACIR BONIFACIO AFFONSO

**LINGUAGEM ELIXIR PARA APRENDIZAGEM DE PROGRAMAÇÃO  
CONCORRENTE**

Trabalho de Conclusão de Curso  
apresentado ao Centro Universitário  
Carioca, como requisito parcial para  
obtenção do grau de Bacharel em  
Ciência da Computação.

Orientador: Prof. DSc. Sérgio Assunção Monteiro

F313I Félix, Claudio

Linguagem elixir para aprendizagem de programação concorrente /  
Claudio Félix, Moacir Bonifacio Affonso. - Rio de Janeiro. 2021.  
59 f.

Orientador: Sérgio Assunção Monteiro

Trabalho de Conclusão de Curso (Graduação em Ciência  
da  
Computação) - Centro Universitário UniCarioca - Rio de  
Janeiro, 2021.

1.Elixir. 2. Erlang. 3. Aprendizagem. 4. Introdução. 5. Programação  
funcional. 6. Programação concorrente I. Affonso, Moacir Bonifacio. II.  
Monteiro, Sérgio Assunção, prof. orient. III. Título.

CDD 005.1

CLAUDIO FÉLIX  
MOACIR BONIFACIO AFFONSO

**LINGUAGEM ELIXIR PARA APRENDIZAGEM DE PROGRAMAÇÃO  
CONCORRENTE**

Trabalho de Conclusão de Curso  
apresentado ao Centro Universitário  
Carioca, como requisito parcial para  
obtenção do grau de Bacharel em  
Ciência da Computação.

**BANCA EXAMINADORA**

---

Prof. Sérgio Assunção Monteiro, DSc.  
Orientador

---

Prof. Alberto Tavares da Silva, DSc.  
Professor Convidado

---

Prof. André Luiz Avelino Sobral, MSc.  
Coordenador

Rio de Janeiro  
2021

## **AGRADECIMENTOS**

Agradecemos a quem esteve ao nosso lado durante toda esta jornada.

Em especial, ao Professor Sérgio Monteiro pela atenção, prestatividade e confiança em nossa capacidade para encerrar esta empreitada.

## RESUMO

Concorrência tende a ser algo de notória complexidade em se adentrar. Muitas das vezes, o processo introdutório constitui-se pelo Paradigma Imperativo, para mais tarde buscar o Concorrente. Nesse sentido, deparar-se com essa nova forma de programar, dado o estabelecimento prévio de uma forma de pensar, tende a ocasionar um atordoamento inicial e, em alguns casos, uma percepção negativa. Diante disso, o presente trabalho tem o objetivo de demonstrar as vantagens do uso de Elixir para a introdução ao paradigma de programação concorrente.

**Palavras-chave:** Elixir. Erlang. Aprendizagem. Introdução. Programação Funcional. Programação Concorrente

## **ABSTRACT**

Concurrence tends to be something of notorious complexity to enter. Often, the introductory process is constituted by the Imperative Paradigm, to later look for the Concurrent. In this sense, coming across this new way of programming, given the previous establishment of a way of thinking, tends to cause an initial dizziness and, in some cases, a negative perception. Therefore, this work aims to demonstrate the advantages of using Elixir to introduce the concurrent programming paradigm.

**Keywords:** Elixir. Erlang. Learning. Introduction. Functional Programming. Concurrent Programming.

## LISTA DE FIGURAS

Figura 1: Exemplo do primeiro interpretador de Erlang.....	16
Figura 2: Sintaxe para recepção de mensagens.....	17
Figura 3: Nó nomeado.....	19
Figura 4: Coleta de lixo geracional.....	21
Figura 5: Supervisors e workers.....	23
Figura 6: Representação abstrata de funcionamento da BEAM.....	24
Figura 7: Funções Puras.....	27
Figura 8: Funções Puras: função valorada.....	27
Figura 9: Funções Alta-Ordem.....	28
Figura 10: Avaliação Preguiçosa.....	29
Figura 11: Aridade.....	29
Figura 12: Recurção de cauda: definição.....	30
Figura 13: Recurção de cauda: processo.....	30
Figura 14: Execução de um processo.....	31
Figura 15: Elixir, compilador de elixir.....	36
Figura 16: IEx: “Hello World”.....	37
Figura 17: Funções: funções anônimas.....	40
Figura 18: Funções: funções nomeadas.....	41
Figura 19: Funções: funções privadas.....	41
Figura 20: Módulos: aninhamento.....	42
Figura 21: Módulos: atributos.....	43
Figura 22: Módulos: defstruct.....	43
Figura 23: Módulos: macro use.....	45
Figura 24: Mix: new.....	45
Figura 25: Mix: estrutura.....	45
Figura 26: Mix: mix.exs.....	46
Figura 27: Mix: deps, função que gere dependências.....	47
Figura 28: Mix: ambiente de produção.....	47
Figura 29: Ping Pong.....	49
Figura 30: Soma 1 + 1 após 10 segundos.....	50
Figura 31: Contador.....	51



Figura 32: Pilha.....	52
Figura 33: Pilha em execução.....	54

## LISTA DE TABELAS

Tabela 1: Propriedades para concorrência.....	15
Tabela 2: Requisitos para Linguagens de Programação de Sistemas de Telecomunicações em switch de comunicação.....	16
Tabela 3: Recepção de Mensagens em Buffer.....	18
Tabela 4: Erlang/OTP.....	25
Tabela 5: Sistema Elixir.....	35
Tabela 6: Tipos Importantes em Elixir.....	37
Tabela 7: Operadores Aritméticos.....	38
Tabela 8: Operadores de Comparação.....	39
Tabela 9: Operadores Lógicos.....	39
Tabela 10: Operadores Importantes.....	40
Tabela 11: Módulos: atributos reservados.....	43
Tabela 12: Diretivas de Módulos.....	44
Tabela 13: Tipos Personalizados.....	44
Tabela 14: Tarefas padrão do Mix.....	46
Tabela 15: Mix: ambientes.....	47
Tabela 16: Tipo de Documentação.....	48

## SUMÁRIO

<b>1. INTRODUÇÃO.....</b>	<b>13</b>
1.1. OBJETIVOS.....	13
1.2. ORGANIZAÇÃO.....	13
<b>2. BASES DE ELIXIR.....</b>	<b>15</b>
2.1. BREVE HISTÓRIA DE ERLANG.....	15
2.2. O INICIO DA LINGUAGEM ERLANG.....	17
<b>2.2.1. Recepção de Mensagens em Buffer.....</b>	<b>17</b>
<b>2.2.2. Tratamento de Erros.....</b>	<b>18</b>
<b>2.2.3. Links.....</b>	<b>18</b>
2.3. ERTS (ERLANG RUN-TIME SYSTEM APPLICATION).....	18
<b>2.3.1. Nó Nomeado.....</b>	<b>19</b>
<b>2.3.2. Compilador Erlang – erlc.....</b>	<b>19</b>
<b>2.3.3. Processos.....</b>	<b>19</b>
2.3.3.1. Bloco de Controle de Processo – PCB (Process Control Block).....	20
2.3.3.2. Estruturas – Heap e Stack.....	20
2.3.3.3. Coletor de Lixo – Garbage Collector (GC).....	20
2.3.3.4. Caixa de Mensagens – Mailbox.....	22
2.3.3.5. Supervisores – Supervisor.....	22
<b>2.3.4. BEAM.....</b>	<b>23</b>
2.4. ERLANG/OTP.....	24
<b>3. PARADIGMAS.....</b>	<b>26</b>
3.1. FUNCIONAL.....	26
<b>3.1.1. O que é uma função?.....</b>	<b>26</b>
<b>3.1.2. Características de Funções.....</b>	<b>27</b>
3.1.2.1. Ordem de Avaliação.....	27
3.1.2.2. Funções Puras (Pure Functions).....	27
3.1.2.3. Transparência Referencial (Refencial Transparency).....	28
3.1.2.4. Funções de Alta Ordem (High-Order Functions).....	28
3.1.2.5. Avaliação Preguiçosa (Lazy Evaluation).....	29
3.1.2.6. Aridade (Arity).....	29
3.1.2.7. Cidadãos de Primeira Classe (First-Class Citizen).....	29
3.1.2.8. Recursão de Cauda (Tail Recursion).....	30

3.1.2.9. Imutabilidade (Immutability).....	30
3.2. CONCORRENTE.....	31
<b>3.2.1. O que é concorrência em computação?.....</b>	<b>31</b>
<b>4. INTRODUÇÃO AO ELIXIR.....</b>	<b>33</b>
4.1. JOSÉ VALIM.....	33
4.2. NASCE A LINGUAGEM ELIXIR.....	34
4.3. OBJETIVOS DE ELIXIR.....	35
4.4. INSTALAÇÃO.....	35
4.5. ELIXIRC.....	36
4.6. IEX.....	36
4.7. TIPOS BÁSICOS.....	37
<b>4.7.1. Átomos.....</b>	<b>37</b>
4.8. TIPOS DE OPERADORES.....	38
4.9. FUNÇÕES.....	40
<b>4.9.1. Funções Anônimas.....</b>	<b>40</b>
<b>4.9.2. Funções Nomeadas.....</b>	<b>41</b>
<b>4.9.3. Funções Privadas.....</b>	<b>41</b>
4.10. MÓDULOS.....	42
<b>4.10.1. Aninhamento de Módulo.....</b>	<b>42</b>
<b>4.10.2. Atributos de Módulos.....</b>	<b>42</b>
<b>4.10.3. Defstructs.....</b>	<b>43</b>
<b>4.10.4. Diretivas.....</b>	<b>44</b>
<b>4.10.5. Tipos Personalizados.....</b>	<b>44</b>
<b>4.10.6. Use.....</b>	<b>44</b>
4.11. MIX.....	45
<b>4.11.1. Gerência de Dependências.....</b>	<b>46</b>
<b>4.11.2. Ambientes do Mix.....</b>	<b>47</b>
4.12. DOCUMENTAÇÃO.....	48
<b>5. ESTUDO DE CASO.....</b>	<b>49</b>
5.1. PING PONG.....	49
5.2. SOMA 1 + 1 APÓS 10 SEGUNDOS.....	50
5.3. CONTADOR.....	51
5.4. PILHA.....	52
<b>6. CONCLUSÃO.....</b>	<b>55</b>

6.1. RECOMENDAÇÕES DE ESTUDOS FUTUROS.....	55
<b>REFERÊNCIAS.....</b>	<b>56</b>

## 1. INTRODUÇÃO

Da telefonia a memes de gatinhos, a usabilidade da internet poderia ser resumida nessa assertiva. Entretanto, dá-se como insuficiente para focar nos artefatos que estruturaram e mantêm muitos dos sistemas até os dias atuais. A partir disso, faz-se interessante investigar as ferramentas que possibilitam a conexão entre as pessoas.

Nesse sentido, uma das mais proeminentes é Erlang. Linguagem desenvolvida de forma a garantir robustez por parte de um sistema. Para que consiga realizar tal ensejo, usufrui de seu poderoso ecossistema, que inclui sua máquina virtual, bibliotecas e padrões de desenvolvimento.

Outra linguagem que também faz uso desse poder, chama-se Elixir. Criada pelo brasileiro José Valim, aproveita muito de Erlang incluindo sua máquina virtual. No entanto, detém de um ferramental próprio, que garante características muito interessantes a quem programa.

### 1.1. OBJETIVOS

Diante do exposto acima, este trabalho tem o objetivo de explorar as potencialidades da linguagem Elixir. Com foco em suas ferramentas, seus paradigmas de programação e sua aplicabilidade mais memorável: concorrência.

### 1.2. ORGANIZAÇÃO

Este trabalho está dividido em seis capítulos. O primeiro, realiza uma breve introdução aos objetivos e apresenta a organização do trabalho.

Já por dentro do segundo capítulo, apresenta-se a linguagem de programação Erlang. Realiza-se uma contextualização histórica para sua criação, uma introdução ao processo histórico de desenvolvimento do ecossistema alastrante, além de um breve mergulho a ele.

Dentro do terceiro capítulo, apresenta-se os paradigmas funcional e concorrente. Dá-se dessa forma, pois a linguagem Elixir é fortemente atrelada a eles e esse ser o foco deste trabalho.

Por sua vez, o quarto capítulo é uma introdução ao Elixir. A história da linguagem, motivações e objetivos que levaram a sua criação são explorados. Além

disso, características e ferramentas principais são apresentadas: IEx, Mix e suas principais funções.

Ao adentrar no quinto capítulo, um estudo de caso ao Elixir. Com ênfase em em aprendizagem, são desenvolvidos exemplos que demonstram as potencialidades de usufruir da máquina virtual do Erlang em conjunto ao paradigmas funcional. Dessa forma, orientamos para a área na qual a linguagem mais se destaca.

Por fim, o sexto capítulo apresenta as conclusão retiradas deste trabalho, além de sugestões para abordagens futuras.

## 2. BASES DE ELIXIR

Antes de adentrar ao Elixir, faz-se necessário mergulhar em seus pilares e conhecer a estrutura que lhe dá suporte. Diante disso, será realizada uma breve apresentação ao Erlang.

### 2.1. BREVE HISTÓRIA DE ERLANG

Ao chegar ao Ericsson Computer Science Laboratory, Joe Armstrong perguntou ao chefe do laboratório, Bjarne Däcker, o que fazer. Recebeu como resposta a seguinte missão: “resolver o problema de software da Ericsson” (Däcker apud Armstrong, 2007, tradução nossa).

Isso foi por volta de 1985, as linguagens de programação convencionais não atendiam às preocupações levantadas: atualização em serviço (in-service) e zero tempo de inatividade (down-time).

Serviços de telefonia tendem a ser altamente interconectados. Uma central telefônica, precisa ter capacidade de lidar com milhares e até milhões de transações em simultaneidade. Diante disso, o serviço não pode sofrer uma pausa para atualização, por exemplo.

Serviços de telefonia também exigem robustez. Em locais mais afastados da matriz, um serviço precisa ser capaz de não passar por processos de recuperação a todo momento. Diante disso, deriva-se a interpretação de um sistema tolerante a falhas.

Para lidar com essas problemáticas, erigiu-se uma necessidade de pensar a partir de alguma dinâmica simultânea. Mike Williams, que introduziu Armstrong a cultura de software da Ericsson, postulou três propriedades para o sucesso de linguagens de programação em operações concorrentes, conforme a Tabela 1:

*Tabela 1: Propriedades para concorrência*

• o momento de criar um processo
• o momento de realizar uma mudança de contexto entre dois processos diferentes
• o tempo para copiar uma mensagem entre dois processos

*Fonte: Williams apud Armstrong, 2007, tradução nossa*



Além disso, outros requisitos também foram levantados. Muitos deles, especificados para linguagens de programação que são voltadas a sistemas de telecomunicações. Apresentadas na Tabela 2, logo abaixo:

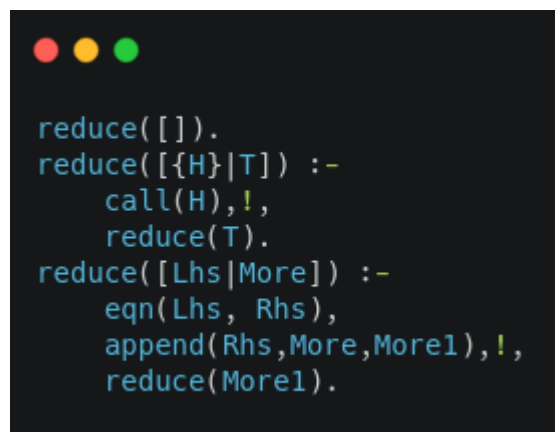
*Tabela 2: Requisitos para Linguagens de Programação de Sistemas de Telecomunicações em switch de comunicação*

• Lidar com um grande número de atividades simultâneas
• Ações a serem realizadas em um determinado ponto do tempo ou dentro um certo tempo
• Sistemas distribuídos em vários computadores
• Interação com hardware
• Sistemas de software muito grandes
• Funcionalidades complexas, como interação de recursos
• Operação contínua ao longo de vários anos
• Manutenção de software (reconfiguração, etc.) sem parar o sistema
• Requisitos rigorosos de qualidade e confiabilidade
• Tolerância a falhas para falhas de hardware e erros de software

*Fonte: Armstrong, 2007, tradução nossa*

Com algumas das bases estabelecidas, Armstrong começou a planejar a lógica da linguagem. Inicialmente, a ideia era estruturar o sistema em SmallTalk. mas após uma conversa com Roger Skagervall mudou para Prolog (Armstrong, 2007, tradução nossa). Com o desenvolvimento, acabou realizando modificações e criou o primeiro interpretador, cujas regras são similares às da Figura 1:

*Figura 1: Exemplo do primeiro interpretador de Erlang*



```

reduce([ ]).
reduce([ {H} | T ]) :-
    call(H), !,
    reduce(T).
reduce([ Lhs | More ]) :-
    eqn(Lhs, Rhs),
    append(Rhs, More, More1), !,
    reduce(More1).

```

*Fonte: adaptado de Armstrong, 2007*

O tempo passou e meu pequeno interpretador cresceu cada vez mais (Armstrong, 2007, tradução nossa). Em 1986, o processo de desenvolvimento ganhou mais notoriedade.

Robert Virding – outro membro do laboratório – se interessou pelo projeto. Nas colocações de Armstrong (2007, tradução nossa), desse encontro surgiram as primeiras formulações do paradigma de programação orientado à concorrência.

Ainda nessa época, o nome para essa nova linguagem foi adotado. O interessante disso, dá-se pelo fato de nenhum dos membros saber exatamente quem nomeou Erlang e nem o que significa. Talvez “Ericsson Language” ou talvez em homenagem Agner Krarup Erlang (1878 – 1929), o curioso é que os próprios autores incentivam tal ambiguidade (Armstrong, 2007, tradução nossa).

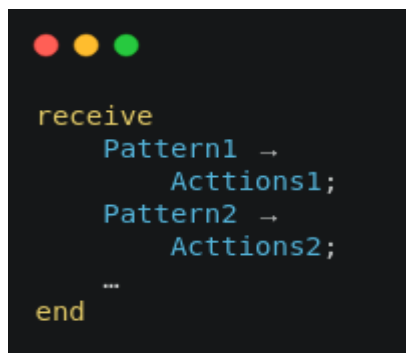
## 2.2. O INICIO DA LINGUAGEM ERLANG

Em 1988, depois de muitas mudanças realizadas, Erlang já tinha sido estabilizada como linguagem. Certas adições são importantes até os dias atuais. Dentre elas, algumas podem ser listadas abaixo:

### 2.2.1. Recepção de Mensagens em Buffer

Erlang adotou em seu design, uma maneira de lidar com trocas de mensagens dentro da linguagem. Através de um formulário de recepção seletiva (buffering selective receive), tem-se o armazenamento da mensagem em uma fila (save queue), caso tal envio não corresponda ao padrão aguardado no recebimento. Dessa forma, dá-se a possibilidade para o tratamento de mensagens fora de ordem.

*Figura 2: Sintaxe para recepção de mensagens*

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in Erlang and shows the 'receive' construct for message reception. It includes patterns and actions separated by arrows, with an 'end' keyword at the bottom.

```
receive
  Pattern1 →
    Actions1;
  Pattern2 →
    Actions2;
  ...
end
```

*Fonte: adaptado de Armstrong,  
2007*

O modo como se lê o funcionamento do código exposto na Figura 2 logo acima, apresenta-se na Tabela 3 logo abaixo:

*Tabela 3: Recepção de Mensagens em Buffer*

• receive	Espere por uma mensagem
• Pattern1 → Actions1;	Se a mensagem corresponder a Pattern1, avalie o código Actions1
• Pattern2 → Actions2;	Se a mensagem corresponder a Pattern2, avalie o código Actions2

*Fonte: adaptado de Armstrong, 2007*

Em continuidade, o buffer entra em ação para tratar as não correspondentes. Não existindo o casamento com nenhum dos padrões (pattern match), a mensagem será ignorada e armazenada numa fila (save queue) para passar por avaliação posterior. Enquanto isso, a instrução de recebimento aguarda a chegada das mensagens que casem com os padrão estipulados.

### **2.2.2. Tratamento de Erros**

Em Erlang, tratar erros diz respeito a dois computadores que se observam. Isso acontece em função de um dos computadores poder sofrer de travamento e não conseguir realizar mais cálculos. Então, a configuração mínima para uma falha no sistema tolerante tem dois computadores (Armstrong, 2007, tradução nossa).

Muito disso advém da ideia de abstrair o hardware. Cada objeto do sistema, só poderá se comunicar por meio de mensagens, não tendo quaisquer informações a fonte de envio. Segundo Armstrong (2007, tradução nossa), a razão para isso era simplificar o modelo de gramática e uniformizar a comunicação entre processos.

### **2.2.3. Links**

De acordo com Armstrong (2007, tradução nossa), a ideia de links adveio de Mike Williams. Por meio de tal característica, os erros podem ser propagados e cada processo pode conhecer o estado doutro com o qual esta vinculado.

## **2.3. ERTS (ERLANG RUN-TIME SYSTEM APPLICATION)**

ERTS, é o sistema para execução de Erlang. Além da linguagem, também é composto por outros mecanismo que permitem ao programa funcionar. Elixir também utiliza esse sistema para sua execução.

### 2.3.1. Nó Nomeado

Ao iniciar uma aplicação alastrada ao Erlang, inicia-se um nó. Que abarca tanto ERTS, quanto a BEAM. Segundo Stenman (2020, tradução nossa), um nó é executado em um processo do sistema operacional.

Mais precisamente, um nó é um sistema de execução que está em execução. De maneira ainda mais precisa, que foi inicializado com um nome, como na Figura 3:

*Figura 3: Nó nomeado*



*Fonte: Autores*

Todas as camadas que constituem um nó, afetam o desempenho da aplicação.

### 2.3.2. Compilador Erlang – erlc

O compilador erlc, compila arquivos com extensão .erl em código de máquina para BEAM. É escrito na linguagem Erlang e compilado por si próprio para código BEAM. Em geral, compõe um grupo de arquivos pré-compilados para o sistema de execução podê-los executar.

### 2.3.3. Processos

Como citado anteriormente, Mike Williams deu muita ênfase aos processos no momento de trabalhar com concorrência. De acordo com LOGAN, MERRITT e CARLSSON (2010, tradução nossa), um processo representa uma atividade contínua; um agente que executa uma parte do código do programa. Isso de maneira concorrente a outros processos, que executam seu próprio código e em seu próprio ritmo.

Em Erlang, os processos são gerenciados pelo escalonador de processos (scheduler). Ao iniciar o ERTS, espaços de endereço compartilhados - similares a

threads, mas conhecidos como CPU scheduler - são erigidos conforme o número de núcleos (cores) da máquina.

Tais schedulers, são responsáveis por pegar os processos a serem executados, lhes atribuindo um número específico de instruções – conhecidas com reduções (reductions).

Cada processo possui sua própria memória de trabalho - uma caixa de mensagens (mailbox), um heap (cujo tamanho inicial é dado em 233 palavras - words) e uma stack (Stenman, 2020, tradução nossa).

Além disso, também possuem um bloco de controle de processo (PCB), que contém informações sobre o processo e sua execução é controlada por um escalonador preemptivo. Cada processo possui um tempo específico para consumo de recursos da máquina.

Em geral, o código como um todo é encapsulado e executado dentro de um processo. Por exemplo a BEAM, máquina virtual de Erlang, roda como um processo.

#### 2.3.3.1. Bloco de Controle de Processo – PCB (Process Control Block)

Nas colocações de Stenman (2020, tradução nossa), contém todos os campos que controlam o comportamento e o estado atual de um processo.

#### 2.3.3.2. Estruturas – Heap e Stack

Como ressaltado, os processos possuem uma pilha (stack) de execução. Ela é usada para controlar a execução, o armazenamento de endereços que retornam passagem de argumentos para funções e manter variáveis locais. Tuplas e listas, são armazenadas na fila (heap).

Heap e Stack são apenas áreas de memória. Stenman (2020, tradução nossa), a pilha (stack) cresce em direção a endereços de memória inferiores e a fila (heap) em direção aos superiores. O que permite impossibilitar um encontro do topo da pilha com o topo da fila, favorecendo ao processo não ficar sem memória e evitando ao máximo a ação do coletor de lixo (garbage collector).

#### 2.3.3.3. Coletor de Lixo – Garbage Collector (GC)

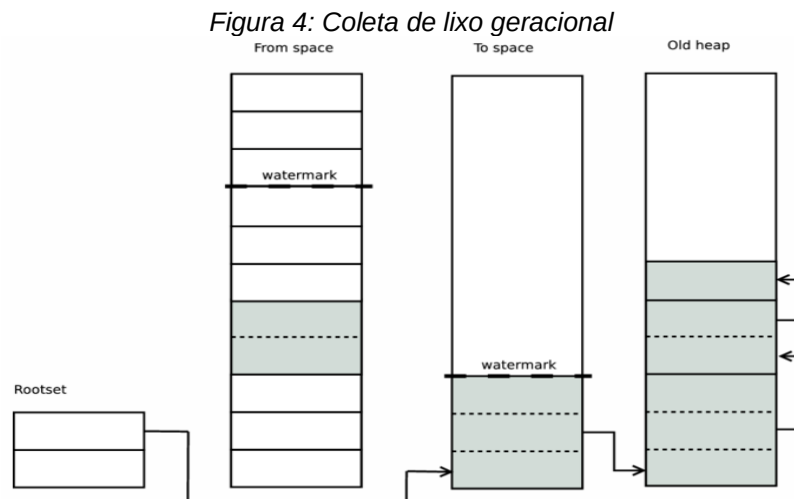
Quando não é possível evitar o encontro entre o topo da pilha (crescente para endereços inferiores) e o topo da fila (crescente para endereços superiores), o

GC entra em ação. Aloca uma nova área de memória chamada de “to space”, percorre a stack a fim de localizar as raízes vivas e copia cada dado dessas raízes encontradas para uma nova heap.

Nas palavras de Stenman (2020, tradução nossa), o GC apenas olha para a parte superior da pilha e move os novos dados para um nomeado young heap, já os antigos vão para um chamado old heap. Em função de ser geracional, usará uma heurística para apenas examinar novos dados, na maioria das vezes.

Note a Figura 4 logo abaixo, retirada da Documentação de Erlang. A partir dela, pode-se observar o processo da coleta de lixo geracional. Alastrada a um old heap, que é usado para armazenar dados de longa duração.

A “watermark”, é uma marca colocada na área em que uma coleta de lixo anterior terminou. Abaixo dela, qualquer coisa é copiada para o old heap durante a fase de cópia. Assim, o coletor de lixo não precisa verificar os termos de longa duração.



Fonte: Documentação Erlang, 2021

Numa próxima passagem do coletor, qualquer apontamento para o old heap é ignorado e a verificação não é realizada. Isso ajuda no desempenho, porque apenas o young heap - o menor - é verificado na maioria das coletas.

#### 2.3.3.4. Caixa de Mensagens – Mailbox

Outra coisa muito importante são as caixas de mensagens, as mailbox. Erlang utiliza um modelo similar ao de atores, através do qual cada processo é um ator, que possui um endereço (PID). O modo como os processos (que são isolados) se comunicam, dá-se pelo envio e recebimento de mensagens de maneira assíncrona.

Um processo envia uma mensagem a outro processo. No momento de chegada dessa mensagem, ela é colocada na mailbox e o processo receptor é agendado para a execução – similar a uma caixa de correio. Em seguida, o que recebeu tenta verificar a correspondência ao padrão da mensagem. Se bem-sucedido, recebe-se a mensagem e sua remoção da mailbox acontece. Se não for bem-sucedido, a mensagem é armazenada na heap.

#### 2.3.3.5. Supervisores – Supervisor

Um dos pontos mais importantes em Erlang, está diretamente relacionado ao projeto de seus recursos. Durante os passos que envolvem o desenvolvimento da linguagem, busca-se por formas de garantir que seja o mais tolerante a falhas possível. Nesse sentido, emergem os supervisores.

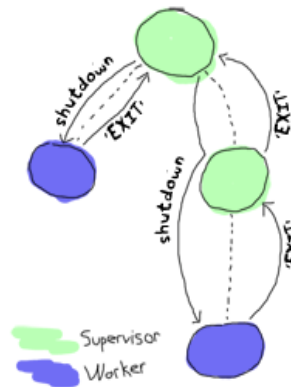
Como citado acima, processos são muito importantes na modelagem de uma linguagem concorrente. Um supervisor é um processo, que supervisiona outros processos nomeados como filhos.

Por sua vez, processos filhos também podem supervisionar outros processos. Ou então, podem ser nomeados como processos de trabalho (worker process). A essa estrutura hierárquica parcamente apresentada, dá-se o nome de árvore de supervisão.

O principal objetivo de um Supervisor, caracteriza-se por monitorar seus processos filhos - gerir seu ciclo de vida. O supervisor pode iniciar ou encerrar um filho. Sua existência é muito útil para a gerência de memória. Desconhecer o estado de um processo, pode ocasionar em um encerramento repentino da máquina virtual.

Na Figura 5, que está logo abaixo, nota-se como funciona uma árvore de supervisores em Erlang e Elixir:

Figura 5: Supervisors e workers



Fonte: Hebert, 2013

#### 2.3.4. BEAM

A máquina virtual do Erlang. Nas colocações de Högberg (2020, tradução nossa), é a máquina de registro, onde todas as instruções operam em registros nomeados. Cada um desses registros pode conter um tipo de dado, um termo (term) de Erlang, como: inteiros, tuplas etc.

Ainda segundo Högberg (2020, tradução nossa), existem dois tipos de registros mais importantes: X e Y.

- X, usado para dados temporários e transmissão de dados entre funções.
- Y, para cada quadro (frame) de pilha.

Não se deve confundir com o ERTS (Erlang Run-Time System Application). BEAM é apenas a máquina virtual, executa as instruções. Não possui noções de processos, portas, tabelas ETS e assim por diante (Högberg, 2020, tradução nossa).

BEAM (Bogdam Erlang Abstract Machine), esse nome advém de seu criador: Bogumil (Bogdan) Hausman. Além dessa máquina, também é criador da TEAM (Turbo Erlang Abstract Machine) e da antiga BEAM, duas máquinas virtuais antecessoras da atual.

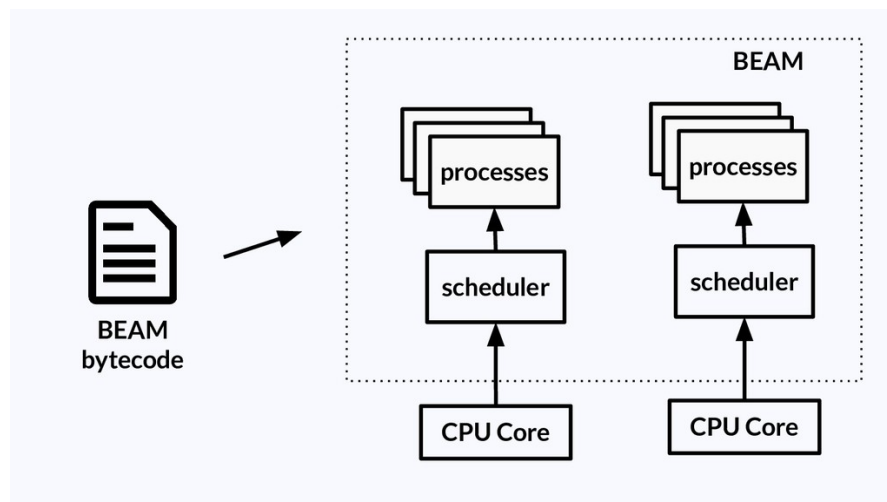


Ambas, por sua vez, sucedem a primeira máquina virtual, JAM (Joe's Erlang Abstract Machine). Criada pelo primeiro desenvolvedor da linguagem Erlang, Joe Armstrong.

Abaixo, a Figura 6 contextualiza o funcionamento da BEAM. Note as setas, elas indicam o prosseguimento de execução até chegar aos processos. Atente-se ainda, para a indicação do bytecode, são os arquivos com extensão .beam.

A máquina virtual (VM), tem-se aqui representada didaticamente. Essa situação, apresentada através de uma abstração de seu funcionamento, sem o contexto do sistema de execução de Erlang. Para compreender com mais afinco, veja 2.3. ERTS (ERLANG RUN-TIME SYSTEM APPLICATION).

Figura 6: Representação abstrata de funcionamento da BEAM



Fonte: Watanabe, 2019

## 2.4. ERLANG/OTP

Em 1990, Claes (Klacke) Wikström chega ao laboratório onde ocorria o desenvolvimento de Erlang. Em 1993, ele faria uma importante adição ao processo de desenvolvimento.

Durante esse período, realizou a introdução da programação distribuída na linguagem – uma ideia antiga, mas que ainda não fora implementada até este momento. Esse foi um passo importante para o que aconteceria em 1997, um ato a fim de buscar a massificação de Erlang.

Nesse mesmo ano, nasce o OTP (Open Telecom Platform). Segundo Armstrong (2007, tradução nossa), é o nome da distribuição e nome da unidade de produto da Ericsson. A Tabela 4 demonstra sua composição:

*Tabela 4: Erlang/OTP*

• Bibliotecas de código Erlang.
• Padrões de Projeto para a construção de aplicativos comuns.
• Documentação.
• Cursos.
• “Como fazer”.

*Fonte: Armstrong, 2007, tradução nossa*

Por meio deste projeto, quem programa utilizando Erlang pode ser capaz de retirar proveito dos anos por dentro do desenvolvimento da linguagem num mesmo ecossistema. Hoje, seu nome é conhecido como Erlang/OTP.

### 3. PARADIGMAS

Nesta seção, são desenvolvidas noções sobre dois dos paradigmas que norteiam as linguagens citadas. Tal ato, dá-se em função de Elixir ser fortemente estruturada sobre o ecossistema de Erlang e ambas usufruírem dos paradigmas citados.

Para reforçar tal escolha de abordagem, de acordo com Armstrong (2007, tradução nossa, p. 6-14, 4.11 BOS – OTP and behaviors),

“Erlang não é uma linguagem funcional estritamente livre de efeitos colaterais, mas uma linguagem concorrente em que o que acontece dentro de um processo é descrito por uma linguagem funcional simples.”

#### 3.1. FUNCIONAL

O paradigma funcional, torna-se cada vez mais atinente fora da academia. As duas linguagens já pontoadas durante o transcorrer deste texto, demonstram sua crescente importância ao desenvolvimento de soluções através de um olhar diferente do convencional.

Diante do exposto, cabe ressaltar a ligação deste paradigma com a matemática. Para tal, primeiramente é importante ressaltar as funções. Elas, constituem grande importância no desenvolvimento dessas linguagens.

##### 3.1.1. O que é uma função?

Dentro da matemática, uma função pode ser estabelecida como um mapeamento entre conjuntos. Esses conjuntos, são conhecidos como domínio e contradomínio.

Tal ligação é realizada, com o intuito de extrair um subconjunto conhecido como conjunto imagem. Esse, podendo ser parte do conjunto contradomínio. Para tanto, tem-se a aplicação de cada elemento do conjunto domínio a um elemento do conjunto contradomínio.

No contexto computacional, uma função pode ser estabelecida como uma rotina. Em geral, que permanece a espera de um ou mais argumentos de entrada e fornece uma saída como resposta dessa entrada.

### 3.1.2. Características de Funções

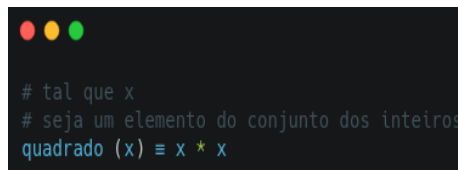
#### 3.1.2.1. Ordem de Avaliação

Uma característica fundamental é que nas funções matemáticas, a ordem de avaliação é controlada por recursão e expressões condicionais, Sebesta (2011). Sem análise sequencial e sem repetição iterativa.

#### 3.1.2.2. Funções Puras (Pure Functions)

Outro ponto marcante é dado pelo seguinte: não sofrem de efeitos colaterais; observe a Figura 7 abaixo:

*Figura 7: Funções Puras*

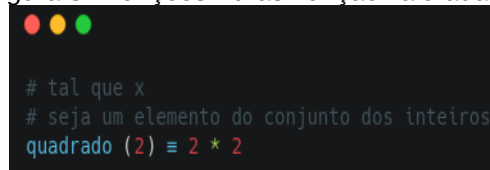


```
# tal que x
# seja um elemento do conjunto dos inteiros
quadrado (x) = x * x
```

*Fonte: Autores*

X, pode ser um parâmetro representativo de qualquer elemento constitutivo do domínio. Entretanto, fixa-se tal elemento para que durante o processo de avaliação da expressão, possa representar um valor específico. Note seu uso com a representação da Figura 8, logo abaixo:

*Figura 8: Funções Puras: função valorada*



```
# tal que x
# seja um elemento do conjunto dos inteiros
quadrado (2) = 2 * 2
```

*Fonte: Autores*

Tais efeitos, estão conectados a variáveis que modelam posições de memória. (Tanto Erlang, quanto Elixir não são linguagens puramente funcionais). Como não possui variáveis, funções não sofrem desses efeitos.

Além disso, funções não mantêm estado. Um valor é definido (como mostrado acima) – não uma sequência de operações em memória – e vinculado uma única vez a uma função (algo parecido com constantes, em linguagens imperativas). A partir dessa definição, pode-se enviar tal valor como argumento para outra função.

### 3.1.2.3. Transparência Referencial (Refencial Transparency)

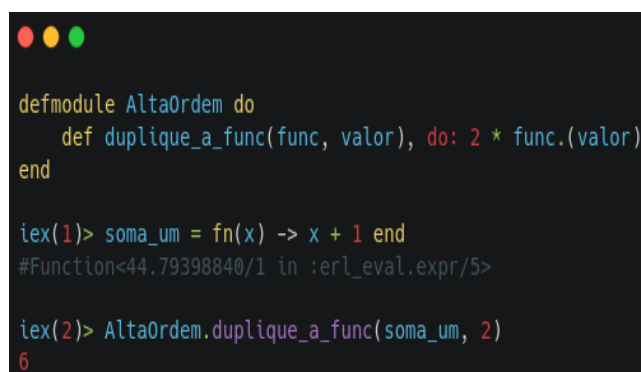
Essa característica combinacional das funções, possui o nome de transparência referencial. No caso do retorno, um mesmo valor é sempre definido na saída, ao ser fornecido o(s) mesmo(s) parâmetro(s) de entrada. O que propicia a sua execução em ordem arbitrária e repetidas vezes, sem que isso leve o programa ao estado de erro.

### 3.1.2.4. Funções de Alta Ordem (High-Order Functions)

Outra característica importante que constitui funções, são as chamadas de funções de alta ordem. Elas podem receber uma ou mais funções ou mesmo retornam uma função.

Note o exemplo em Elixir na Figura 9, demonstrando a definição e execução da função `duplique_a_func`, que recebe `soma_um` e dobra seu valor:

*Figura 9: Funções Alta-Ordem*



```
defmodule AltaOrdem do
  def duplique_a_func(func, valor), do: 2 * func.(valor)
end

iex(1)> soma_um = fn(x) -> x + 1 end
#Function<44.79398840/1 in :erl_eval.expr/5>

iex(2)> AltaOrdem.duplique_a_func(soma_um, 2)
6
```

*Fonte: adaptado de Laurent e Eisenberg, 2016*

A execução do código apresentado, dá-se por meio do IEx (veja 4.6 IEX). Isso por meio da importação do módulo, num arquivo com extensão `.ex` ou mesmo `.beam`, que é compilado (veja a seção 4.5. ELIXIRC). Ou por meio de sua definição dentro do próprio IEx.

### 3.1.2.5. Avaliação Preguiçosa (Lazy Evaluation)

Avaliação preguiçosa significa que um parâmetro será avaliado apenas no momento em que seu valor for necessário. Ajuda a reduzir a complexidade de tempo em linguagens funcionais.

Na linguagem Elixir, o módulo Stream comporta funções para realizar essa operação. Uma delas é a map, que recebe uma coleção de enumeráveis e uma função. A função recebida terá sua aplicação feita, a cada elemento dos enumeráveis enviados. Na Figura 10 listada abaixo, a saída terá a exibição dos dez primeiros números:

Figura 10: Avaliação Preguiçosa



```
iex(1)> Stream.map(1..20_000_000, &(&1)) |> Enum.take(10)
```

Fonte: adaptado de Davi, 2017

### 3.1.2.6. Aridade (Arity)

Aridade, esta correlacionada ao número de argumentos que uma função pode receber. Em Elixir, a aridade é representada por uma barra e um valor, colocadas após o nome da função. Note a Figura 11:

Figura 11: Aridade



!=/2	%/2
%{}/1	&&/2
&/1	*/2
++/2	+/1
+/2	--/2
-/1	-/2

Fonte: Autores

### 3.1.2.7. Cidadãos de Primeira Classe (First-Class Citizen)

Funções que são capazes de se comportar como qualquer outro tipo de objeto, são conhecidas como cidadãos de primeira classe.

### 3.1.2.8. Recursão de Cauda (Tail Recursion)

Como programação funcional faz uso de recursão, isso pode significar lentidão em alguns casos. Para minimizar tal acontecimento, linguagens funcionais utilizam recursão de cauda como solução parcial para essa problemática.

No raciocínio de Sebesta (2011), se uma chamada recursiva é a última expressão na função, ela é chamada de recursão de cauda. Observe as Figuras 12 e 13, definição e execução da recursão de cauda:

Figura 12: Recursão de cauda: definição

```
defmodule FatorialComRecursaoDeCauda do
  def de(valor), do: fatorial_de(valor, 1)
  defp fatorial_de(0, acumulador), do: acumulador
  defp fatorial_de(valor, acumulador) when valor > 0, do: fatorial_de(valor - 1, valor * acumulador)
end

FatorialComRecursaoDeCauda.de 3
```

Fonte: adaptado de Almeida, 2018

Figura 13: Recursão de cauda: processo

```
FatorialComRecusaoDeCauda.fatorial_de 3, 1
→ FatorialComRecusaoDeCauda.fatorial_de 2, 3
→ FatorialComRecusaoDeCauda.fatorial_de 1, 6
→ FatorialComRecusaoDeCauda.fatorial_de 0, 6
```

Fonte: Autores

### 3.1.2.9. Imutabilidade (Imutability)

Propriedade de uma variável imutável, que simula uma incógnita matemática (Manzano, 2021). Em miúdos, ao se definir um valor em uma posição de memória, esse não sofrerá alteração alguma durante o trabalho de processamento do programa.

Mesmo assim, pode-se realizar a “troca” de tal valor. No momento em que se fizer necessário tal ato, um “novo” valor será definido em uma instância de memória diferente do original. Dessa forma, possibilitando a preservação do inicialmente definido e a utilização do posteriormente definido.

### 3.2. CONCORRENTE

Como é parte integrante do ecossistema Erlang, que por sua vez é a base de Elixir, a programação concorrente tem sua importância aqui ressaltada. Diante disso, apresenta-se como fulcral compor parcialmente com o que corresponde a isso no contexto da computação.

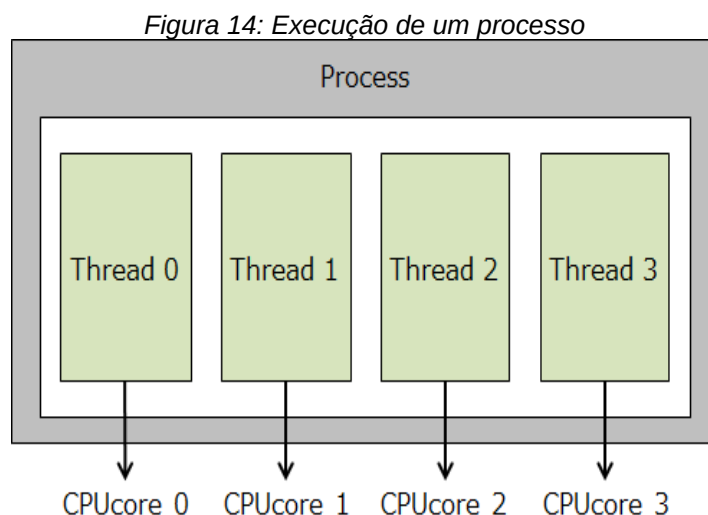
#### 3.2.1. O que é concorrência em computação?

Programação concorrente é nome dado a técnicas de programação para resolver os problemas de sincronização e comunicação (Ben-Ari, 1982, tradução nossa), entre tarefas (tasks) computacionais.

Sob esse modo de desenvolver, são construídos programas para realizar várias tarefas simultâneas. Essas, podendo ter sua execução em um ou vários processadores. Cujas gerência é realizada por algum escalonador de processos (scheduler).

O balanceamento é realizado, para garantir o melhor uso do trabalho de processamento. Com cada tarefa (task), retirando proveito do processador num instante por vez. Uma forma de otimizar sua execução.

Observe a Figura 14, por meio dela se pode reforçar o exposto acima. Na imagem, cada núcleo (core) do processador, comporta uma thread – uma linha de execução de um processo – que pode exercer parte do trabalho de processamento.



*Fonte: StackOverflow, 2015*



Para tal ato, como já citado, Elixir usufrui da base de Erlang. De seu poderoso sistema de execução, suas bibliotecas e todo modo Erlang de realizar as atividades. Contudo, Elixir também traz ferramentas muito interessantes que compõe ainda mais todo ecossistema.

## 4. INTRODUÇÃO AO ELIXIR

Elixir. Esse é o nome a ser abordado ao longo deste capítulo. Nome esse, que o próprio Criador não possui muita clareza ao rastrear de onde saiu; algo muito parecido com o nome Erlang.

Aqui, apresenta-se o Criador e também sua Criação. O que o levou a criar essa linguagem, quais os problemas enfrentados e as soluções descobertas. Isso, para uma melhor compreensão do escopo abordado.

### 4.1. JOSÉ VALIM

Nascido em Porto Alegre, Rio Grande do Sul, Brasil. Teve sua vida construída dentro do estado de Goiás. Já sua vida acadêmica, no entanto, foi aprofundar no estado de São Paulo.

Aos 17 anos, adentrou à Universidade de São Paulo (USP) para cursar Engenharia da Computação. Uma vez ali, suas primeiras linha de código foram tecidas usando a linguagem C. Contudo, seu desejo por programar surgiu num hiato entre os períodos acadêmicos.

Durante as férias, no ano de 2004, Valim decidiu desfrutar do ato de programar. Utilizando ActionScript, construiu para a banda da qual fazia parte – formada junto aos amigos – um site. Com muitas animações, como era de costume ao usar a plataforma Flash dentro do contexto de desenvolvimento WEB.

Já ao final de 2006, aprendeu a linguagem Ruby. De modo gradativo, foi se entremeando ao seio comunitário dessa linguagem, o que lhe garantiu um conforto maior para divulgar seu primeiro projeto de código aberto (open source), por volta de 2007 ou 2008.

Durante essas andanças, se tornou core committer de “Ruby on Rails”. Como desenvolvedor dessa ferramenta, se deparou com discussões a respeito de concorrência. Mais precisamente, sobre a forma de utilização dos núcleos (cores) da máquina.

Em linguagens como Ruby, a utilização de todos os núcleos (cores) da máquina dá-se por meio de threads linhas de execução. Ruby on Rails garantia ser thread safe. Nesse sentido, o software rodando não iria sofrer problemas

relacionados ao uso de threads, como tentar mudar um dado em memória, num espaço compartilhado, ao mesmo tempo.

Contudo, isso não fazia sentido para Valim. Ao notar que a solução de Ruby on Rails não detinha uma resposta suficiente para a sua problemática, decidiu procurar outras formas de como resolvê-lo.

Nessa busca, dois pontos foram fulcrais para o não retorno ao Ruby: imutabilidade e distribuição. Como já citado anteriormente, Erlang possui ambas as características. No entanto, a concorrência ainda era um ponto chave a ser estudado no contexto da problemática.

Durante essa absorvência informacional, deparou-se com o livro *Seven Languages in Seven Weeks*, de Bruce Tate. Por dentro deste, a concorrência é apresentada como uma das perspectivas de comparação entre linguagens ali estudadas. Isso o ajudou a consolidar sua escolha por Erlang em função de sanar a problemática.

Contudo, a linguagem não o satisfazia plenamente. Após explorar o ecossistema e outros sabores – como LFE (LISP Flavour Erlang), chegou a conclusão que precisaria de algo novo. Ao amar o que encontrara e odiar o que faltara, Valim decidiu imbuir ao Erlang a poção que desejava.

#### 4.2. NASCE A LINGUAGEM ELIXIR

Em 2011, depois de muito estudo e desenvolvimento de suas habilidades, Valim realiza o primeiro commit da linguagem Elixir. Um protótipo inicial, um tanto diferente do modelo no qual se encontra atualmente. Contudo, o início já fora dado e um empurrão a mais era necessário.

Assim, por volta de 2012, levou seu projeto individual para dentro da empresa na qual trabalhava, a Plataformatec. Lá, explicou o desejo em desenvolver, graduando as potencialidades da base alastrante, em função do ensejo que obtivera. Um projeto de pesquisa que poderia não dar em nada, mas esse não foi o caso.

A versão 1.0.0 da linguagem, saiu em 10 de setembro de 2014. Durante o período, os desenvolvedores entraram em contato com pessoas interessadas em escrever, falar e usar Elixir. Isso lhes sinalizou, sobre a possibilidade de fundamentos sólidos para o futuro dessa nova linguagem.

#### 4.3. OBJETIVOS DE ELIXIR

Para ser mais preciso, alguns desses fundamentos dizem respeito aos seus objetivos. Elixir tem uma base muito poderosa, que herdou de Erlang. Contudo, não se resume apenas isso.

Ao ser construída, Valim pensou em agregar com o conhecimento que detinha. Em Ruby, a metaprogramação é algo muito forte e motivo de leve frustração pela não similaridade em Erlang. Daí, surge um dos pilares para criação de Elixir: a produtividade.

Contudo, isso não era o suficiente para Valim. Seu ensejo em usufruir ao máximo do ecossistema de Erlang, o levou a construir uma linguagem extremamente integrada. Outro pilar de Elixir é erigido: compatibilidade.

Além disso, Valim estava engajado em ampliar as possibilidades de lidar com a linguagem. Em sua visão, ao tratar de polimorfismo, uma estrutura que implemente um contrato e tenha sua execução realizada faria muito bem ao desenvolvimento. Mais um pilar de Elixir se ergue: extensibilidade.

#### 4.4. INSTALAÇÃO

Antes de instalar a linguagem é válido salientar que, durante a conjuntura de realização deste projeto, Elixir é mantida sob a Apache License 2.0. Uma licença permissiva que requer a preservação de direitos autorais (copyright), avisos de licença e está compatível com a versão 3 da GNU GPL.

Outro ponto importante é que, ao realizar sua instalação, também são integrados os elementos listados na Tabela 5, logo abaixo:

*Tabela 5: Sistema Elixir*

• Console de comando
• Sistema de logger
• Estrutura (framework) de testes
• Linguagem de modelos (templates) simples
• Construtor de código
• A linguagem Elixir

*Fonte: Autores*

Diante disso, recomenda-se acessar ao site oficial da linguagem Elixir, para realizar sua instalação <<https://elixir-lang.org/install.html>>. Abaixo, alguns elementos de Elixir são apresentados.

#### 4.5. ELIXIRC

Compilador de Elixir, que compila binários com a extensão .ex. A partir dos arquivos compilados, são gerados arquivos com a extensão .beam. Tais arquivos podem ser executados ao abrir o IEx, no mesmo local onde se encontram. Apenas, realizando a chamada do módulo.

*Figura 15: Elixirc, compilador de elixir*



*Fonte: Autores*

Já o arquivo binário de texto com a extensão .exs, caracteriza-se como um código para ser executado sem prévia compilação. Logo acima, a Figura 15 apresenta o comando para a compilação de um arquivo Elixir com seu compilador.

#### 4.6. IEX

IEx, é um console de comando onde são digitados códigos da linguagem Elixir e os resultados são obtidos no instante seguinte. Nos permite realiza a avaliação de expressões. Para sua execução, basta num console digitar iex e apertar a tecla enter.

IEx também é um REPL (Read-Eval-Print-Loop), um loop que interpreta e executa códigos diretamente no terminal.

O número do IEx(n) representa o número de execuções bem sucedidas, sendo incrementado a cada execução. Ele sempre é zerado quando iniciado em uma nova execução do terminal. Veja a Figura 16 logo abaixo:

Figura 16: IEx: "Hello World"



Fonte: Autores

## 4.7. TIPOS BÁSICOS

Elixir possui alguns tipos básicos, os quais são necessário para qualquer que deseje adentrar a seu ecossistema. A Tabela 6 os apresenta neste documento.

Uma coisa é importante ressaltar, um termo (term) é qualquer valor (any) em Elixir, assim como em Erlang.

Tabela 6: Tipos Importantes em Elixir

Term	Data Type
• 10	Integer
• 10.1	Float
• A, a, :ok, nil, true	Atom
• "a"	BitStrings
• [], 'a'	List
• {}	Tuple
• %{}	Map
• fn → end	Function

Fonte: Autores

### 4.7.1. Átomos

Átomos são constantes os quais tem em seu nome, seu próprio valor. Também podem ser classificadas como símbolos. São representadas com dois pontos (:) no início do nome.

Não são mutáveis e nem mesmo coletados pelo coletor de lixo (garbage collector). Dessa forma, perduram em memória até o termino de execução do programa. Observe algo importante: true e false são átomos, usados para operações booleanas.

#### 4.8. TIPOS DE OPERADORES

Elixir possui tipos de operadores que podem ser agrupados e categorizados. Algumas associações básicas podem ser realizadas, para direcionar a rápida aprendizagem do uso da linguagem. Abaixo, uma composição de tabelas mostra um pouco disso.

Antes, faz-se necessário citar cada tabela. A Tabela 7, apresenta alguns operadores aritméticos utilizados. Já a Tabela 8, demonstra alguns operadores de comparação que também podem ser utilizados. A Tabela 9 lista os operadores lógicos. Por fim, a Tabela 10 agrega operadores importantes que aparecem bastante em Elixir.

*Tabela 7: Operadores Aritméticos*

Aritméticos	Definição
• +	Adição de termos
• -	Subtração de termos
• *	Multiplicação de termos
• /	Divisão de termos
• div()	Função usada para o quociente da divisão
• rem()	Função usada para o resto da divisão

*Fonte: Autores*

Tabela 8: Operadores de Comparação

Comparação	Definição
• ==	Da esquerda é igual ao da direita
• !=	Da esquerda é diferente ao da direita
• ===	Da esquerda é igual ao tipo da direita
• !==	Da esquerda é diferente ao tipo da direita
• >	Da esquerda é maior que o da direita
• <	Da esquerda é menor que o da direita
• >=	Da esquerda é maior ou igual que o da direita
• <=	Da esquerda é menor ou igual que o da direita
• =	Verifica o casamento do lado esquerdo com o direito

Fonte: Autores

Tabela 9: Operadores Lógicos

Lógicos	Definição
• and	Esquerda e direita são verdadeiros
• or	Esquerda ou direita são verdadeiros
• not	Nega o elemento
• &&	Similar ao and, mas o da esquerda não precisa ser um booleano
•	Similar ao ou, mas o da esquerda não precisa ser um booleano
• !	Similar ao not, mas não precisa ser um booleano

Fonte: Autores



Tabela 10: Operadores Importantes

Importantes	Definição	Exemplo
>	Pipe Operator que permite canalizar comandos	"ola"  > String.length
<>	Concatenação de strings	"Olá," <> " mundo!"
&	Taquigrafia para realizar funções anônimas	multi = &(&1 * &2)
.	Chamada de funções anônimas com passagem de valor	multi.(2, 3)
when	Guarda, cláusula que permite especializar uma função	multi = fn(x, y) when x > y → x * y end
++	Concatenação de listas	[1, 2, 3] ++ [4]
--	Subtração de listas	[1, 2, 3, 4] -- [4]

Fonte: Autores

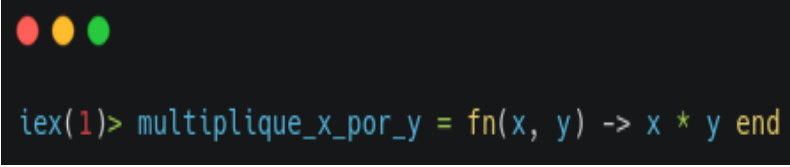
## 4.9. FUNÇÕES

Elixir é uma linguagem que utiliza fortemente as funções. Com elas, a linguagem utiliza casamento de padrões (pattern matching) entre os argumentos da chamada e sua aridade.

### 4.9.1. Funções Anônimas

Como o nome já contextualiza, as funções anônimas são as que não possuem nome. Podem ser passadas para outras funções ou atribuídas (bind) para a uma variável. Veja a Figura 17 logo abaixo:

Figura 17: Funções: funções anônimas



```
iex(1)> multiplique_x_por_y = fn(x, y) -> x * y end
```

Fonte: Autores

### 4.9.2. Funções Nomeadas

São funções que recebem um nome. São sempre definidas dentro de um módulo e podem ficar disponíveis para uso de outros módulo, quando não estão privadas. Observe a Figura 18 logo abaixo:

Figura 18: Funções: funções nomeadas

```
defmodule Exiba do
  def ola(você \\ "mundo"), do: IO.puts "Olá, #{você}"
end

iex(1)> Exiba.ola "Luffy"
Olá, Luffy!

iex(2)> Exiba.ola
Olá, mundo!
```

Fonte: Autores

Um aspecto importante a ressaltar, demonstra-se pelos argumentos da função `ola()`. As duas barras (invertidas), separam o parâmetro que receberá o argumento esperado, do argumento padrão da função. Na primeira chamada, uma `BitString` (veja a Tabela 6) é enviada e na segunda, `ola()` retorna sua resposta padrão.

### 4.9.3. Funções Privadas

Como definido acima, caso não queira outro módulo usufruindo de sua função, pode-se defini-la como privada ao utilizar a macro `defp`. Veja a Figura 19:

Figura 19: Funções: funções privadas

```
defmodule MeDiga do
  defp frase_privada, do: ", você é um bundão!"
  def meu_nome_e_algo_mais(seu_nome), do: seu_nome <> frase_privada
end

MeDiga.meu_nome_e_algo_mais("Bardo")
```

Fonte: Autores

## 4.10. MÓDULOS

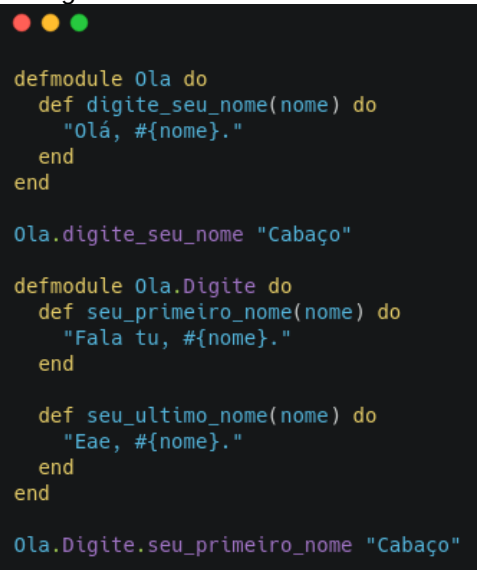
Como já demonstrado acima em diversos momentos, módulos permitem que se organize as funções em espaços de nome (namespaces). Além de possibilitar a definição de função nomeadas e privadas.

### 4.10.1. Aninhamento de Módulo

Também é possível aninhar módulos, em busca de poder tê-los em um espaço de nome (namespace) e usar suas funcionalidades.

Mais abaixo, a Figura 20 ilustra tal aninhamento. Por meio dela, pode-se observar que os dois módulos constituem uma similaridade, com o `Ola.Digite` sendo uma chamada estendida de `Ola`.

Figura 20: Módulos: aninhamento



```
defmodule Ola do
  def digite_seu_nome(nome) do
    "Olá, #{nome}."
  end
end

Ola.digite_seu_nome "Cabaço"

defmodule Ola.Digite do
  def seu_primeiro_nome(nome) do
    "Fala tu, #{nome}."
  end

  def seu_ultimo_nome(nome) do
    "Eae, #{nome}."
  end
end

Ola.Digite.seu_primeiro_nome "Cabaço"
```

Fonte: Autores

### 4.10.2. Atributos de Módulos

Muito parecidos com constantes, podem ser definidos na parte superior do módulo e usados ao seu decorrer. Veja a Figura 21:

Figura 21: Módulos: atributos

```
defmodule Digite do
  @beba "Beba"; @agua "água"
  def seu_nome(nome) do
    ~s("#{@beba} #{@agua}, #{nome}.")
  end
end
Digite.seu_nome "água"
```

Fonte: Autores

Elixir possui atributos reservados, veja a Tabela 11 abaixo:

Tabela 11: Módulos: atributos reservados

• @moduledoc	Documenta o módulo atual
• @doc	Documenta a função ao macro
• @typedoc	Documenta tipos personalizados
• @behavior	Usa um OTP ou comportamento definido

Fonte: Autores

#### 4.10.3. Defstructs

São caracterizadas como um conjunto definido de chaves e valores padrões, mapeamentos especiais. Definidas dentro de um módulo.

Figura 22: Módulos: defstruct

```
defmodule MinhaStruct do
  defstruct nome: "Klebinho", caracteristicas: [:da_rima]
end

# %MinhaStruct{caracteristicas: [:da_rima], nome: "Klebinho"}
%MinhaStruct{}

# %MinhaStruct{caracteristicas: [:devil], nome: "Kazuya"}
kazuya = %MinhaStruct{nome: "Kazuya", caracteristicas: [:devil]}

# %MinhaStruct{caracteristicas: [:devil], nome: "Kazuya Mishima"}
kazuya = %{kazuya | nome: "Kazuya Mishima"}
```

Fonte: Autores

Note a sintaxe começada pelo caractere % na Figura 22. Assim, pode-se adicionar ou atualizar um valor referente ao que foi definido inicialmente.

#### 4.10.4. Diretivas

Elixir possui diretivas para uso dentro dos módulos. Em geral, são definidas na parte superior dos módulos, como os atributos antes citados. A Tabela 12 apresenta as diretivas mais utilizadas.

*Tabela 12: Diretivas de Módulos*

Diretiva	Definição	Exemplo
alias	Possibilita chamar nome por um apelido	alias List, :as Lista
import	Permite importar módulos	import List, only: [first: 1]
require	Importa apenas macros	require SuperMacros

*Fonte: Autores*

#### 4.10.5. Tipos Personalizados

Através de seus pilares, Elixir permite que adaptações sejam realizadas em sua estrutura inicial dos módulos, abaixo algumas diretivas para realizar tal ensejo são listadas na Tabela 13:

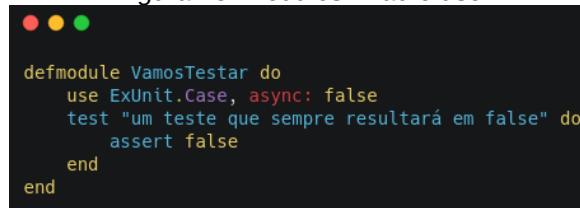
*Tabela 13: Tipos Personalizados*

Anotação	Definição	Exemplos
@spec	Tipo de entrada e saída	@spec minha_func(integer) :: integer
@type	Tipo e estrutura interna são públicos	@type t :: integer
@typep	Tipo é privado e só pode ser usado dentro do módulo no qual foi definido	@typep t :: string
@opaque	Tipo é público, mas estrutura interna é privada	@opaque t :: %MinhaStruct{nome: string, caracteristica: list}

*Fonte: Autores*

#### 4.10.6. Use

A macro “use”, permite atribuir ao módulo habilidades para modificar outro módulo. A Figura 23 mostra sua utilização, para adaptar a estrutura (framework) ExUnit para a realização de testes unitários.

*Figura 23: Módulos: macro use*

```
defmodule VamosTestar do
  use ExUnit.Case, async: false
  test "um teste que sempre resultará em false" do
    assert false
  end
end
```

*Fonte: Autores*

#### 4.11. MIX

Umas das ferramentas mais importantes de Elixir. Pode-se colocar como um construtor de código, uma ferramenta de compilação. Desenvolvida para fornecer tarefas (tasks) capazes de criar, compilar e testar projetos Elixir.

Sua base é um projeto. Em geral, define-se dentro de um módulo cujo nome é mix.exs. Atente-se para a extensão .exs, um arquivo interpretado.

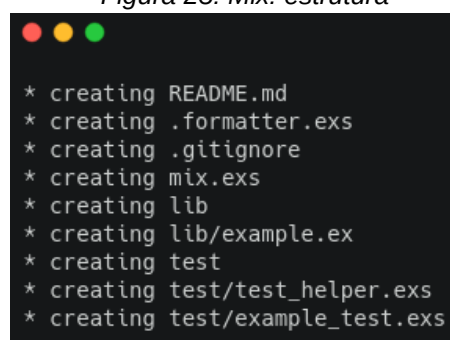
A melhor forma de começar a usufruir de seus poderes, dá-se pela chamada do Mix através da linha de comando, note a Figura 24:

*Figura 24: Mix: new*

```
$ mix new meu_projeto
```

*Fonte: Autores*

Depois do comando, alguns arquivos são gerados. Veja a Figura 25:

*Figura 25: Mix: estrutura*

```
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/example.ex
* creating test
* creating test/test_helper.exs
* creating test/example_test.exs
```

*Fonte: Autores*

Dentre esses arquivos, está o `mix.exs`. Este, que é diretamente responsável pelo armazenamento das configurações da aplicação. Contidas nele estão as dependência, ambiente e versão.

A representação da Figura 26, logo abaixo, está incompleta por questões de visualização. Contudo, nota-se a definição da macro `use` para estender o `Mix.Project` módulo e a declaração da função `project`. Essa, que recebe uma lista contendo o nome da aplicação e sua versão:

Figura 26: Mix: `mix.exs`

```
defmodule MyApp.MixProject do
  use Mix.Project

  def project do
    [
      app: :my_app,
      version: "1.0.0"
    ]
  end
end
```

Fonte: HexDocs, 2021

Com o projeto definido, um conjunto de tarefas que compõe o padrão tem sua possibilidade de execução pela linha de comando, atente-se para a Tabela 14 logo abaixo:

Tabela 14: Tarefas padrão do Mix

• <code>mix compile</code>	Compila o projeto
• <code>mix test</code>	Executa os testes
• <code>mix run</code>	Executa o projeto

Fonte: adaptado de HexDocs, 2021

#### 4.11.1. Gerência de Dependências

Com Mix é possível gerenciar dependências. Dentro do arquivo “`mix.exs`”, existe uma seção chamada de “`deps`”, uma função na verdade. Composta por uma lista e por uma ou mais tuplas.

As tuplas possuem um nome do pacote, constituído por um átomo (atom). Já os valores entre aspas duplas (que são do tipo BitString), dizem respeito a qual versão será utilizada como dependência.

Figura 27: Mix: `deps`, função que gere dependências

```
def deps do
  [
    {:phoenix, "~> 1.1 or ~> 1.2"},
    {:phoenix_html, "~> 2.3"},
    {:cowboy, "~> 1.0", only: [:dev, :test]},
    {:slime, "~> 0.14"}
  ]
end
```

Fonte: Elixir School, 2021

A Figura 27 mostra que a dependência `:cowboy`, só tem sua utilização no contexto de desenvolvimento ou de teste.

#### 4.11.2. Ambientes do Mix

Mix tem o suporte para ambientes diferentes. Isso garante flexibilidade ao momento de desenvolver, pois permite qualificar o código em relação ao contexto. Veja a Tabela 15, apresenta tais ambientes:

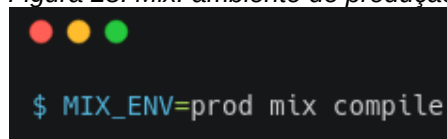
Tabela 15: Mix: ambientes

• <code>:dev</code>	Ambiente padrão
• <code>:test</code>	Contexto de testes
• <code>:prod</code>	Contexto de produção

Fonte: Autores

O ambiente pode ser alterado ao utilizar uma variável de ambiente, veja na Figura 28:

Figura 28: Mix: ambiente de produção



```
$ MIX_ENV=prod mix compile
```

Fonte: Autores



#### 4.12. DOCUMENTAÇÃO

Em Elixir, sua documentação é tratada como uma cidadã de primeira classe. Dessa maneira, são oferecidas várias funções de acesso e geração de documentação para seus projetos. O núcleo da linguagem, fornece muitos atributos diferentes para anotar uma base de código, veja a Tabela 16 logo abaixo:

Tabela 16: Tipo de Documentação

Anotações	Definição	Exemplo
#	Utiliza o caractere conhecido como tralha ou hash (#), antes de alguma construção em linha.	# Lista de elementos. lista = [1, "a", :ok]
@moduledoc	Descrições do que um módulo comporta, o que se pode fazer ao utilizá-lo e de suas funções	@moduledoc """ Provê a função soma_um/1. """
@doc	Descrições do que uma função comporta, o que se pode fazer ao utilizá-la	@doc """ Soma a um ao número enviado """
@typedoc	Descrições do que um tipo comporta, o que se pode fazer ao utilizá-lo	@typedoc """ Tipo que representa um inteiro. """

Fonte: Autores

## 5. ESTUDO DE CASO

Elixir possui em sua base, o favorecimento do trabalho com programação concorrente. Diante disso, exemplos voltados a esse paradigma têm sua exploração realizada aqui. Com o intuito principal em reforçar o uso da linguagem para isso.

### 5.1. PING PONG

O primeiro exemplo espera receber um `:ping` para devolver um `:pong`. O interessante da Figura 29, dá-se pela associação da função `self()` - que retorna o PID (veja a seção 2.3.3. Processos) do processo atual - ao nome `pid_do_processo_pai`. Isso permite que o PID do IEx (veja seção 4.6. IEX) possa ser armazenado sem a necessidade de chamar `self()` novamente.

O que acaba tendo importância na criação de um novo processo. Na linha iniciada pelo nome `pid_do_processo_filho`, a função `spawn()` recebe uma função anônima (veja 4.9.1. Funções Anônimas) justamente com esse intuito. Dentro dela, a função `send()` vai receber o PID do processo que receberá uma mensagem, além da própria mensagem a ser enviada, aqui representada por uma tupla.

Figura 29: Ping Pong



```
iex(1)> pid_do_processo_pai = self()
#PID<0.106.0>

iex(2)> pid_do_processo_filho = spawn fn -> send(pid_do_processo_pai, {:ping, pid_do_processo_pai}) end
#PID<0.109.0>

iex(3)> receive do
... (3)>   {:ping, pid_do_processo_pai} ->
... (3)>   {:pong, pid_do_processo_pai, pid_do_processo_filho}
... (3)> end
{:pong, #PID<0.106.0>, #PID<0.116.0>}
```

Fonte: Autores

Note a macro `receive` (veja a seção 2.2.1. Recepção de Mensagens em Buffer). A partir dela, inicia-se uma estrutura que permite a um processo esperar uma mensagem. Na linha logo abaixo, uma tupla formada por um `:ping` e o termo `pid_do_processo_pai` são usados para buscar o casamento dos padrões (pattern

matching) com a mensagem. Caso haja tal casamento, uma tupla contendo um `:pong` e o termo `pid_do_processo_filho` são retornados.

Como citado em 2.3.3. Processos, cada processo é isolado e só se comunica por troca de mensagens. Similares a threads, são mais simples e tem sua execução realizada em todas as CPUs da máquina. O que torna possível começar a interpretar o motivo pelo qual são bem leves.

## 5.2. SOMA 1 + 1 APÓS 10 SEGUNDOS

Ainda nessa linha de simplicidade, Elixir possui abstrações para trabalhar com processos. Daí surgem as Tasks, convenientemente usadas para computar código sequencial de maneira concorrente. Isso, tendo a possibilidade de executar um código em segundo plano.

*Figura 30: Soma 1 + 1 após 10 segundos*



```
iex(1)> sem_task = fn -> :timer.sleep(10 * 1000) ; 1 + 1 end
#Function<45.65746770/0 in :erl_eval.expr/5>

iex(2)> sem_task.() # Trava o IEx durante 10 segundos
2

iex(3)> com_task = Task.async(fn -> :timer.sleep(10 * 1000) ; 1 + 1 end)
%Task{
  owner: #PID<0.106.0>,
  pid: #PID<0.109.0>,
  ref: #Reference<0.2475943324.1545338882.80312>
}

iex(4)> Task.await(com_task)
2
```

*Fonte: Autores*

A Figura 30 mostra um código que roda dentro do IEx, seu processo pai. Tanto o termo `sem_task` quanto o termo `com_task`, associam-se a funções anônimas, que são constituídas por uma função `:timer.sleep()` - permite adormecer o sistema – e a soma de um mais um. Contudo, o uso de Task permite otimizar a execução de código.

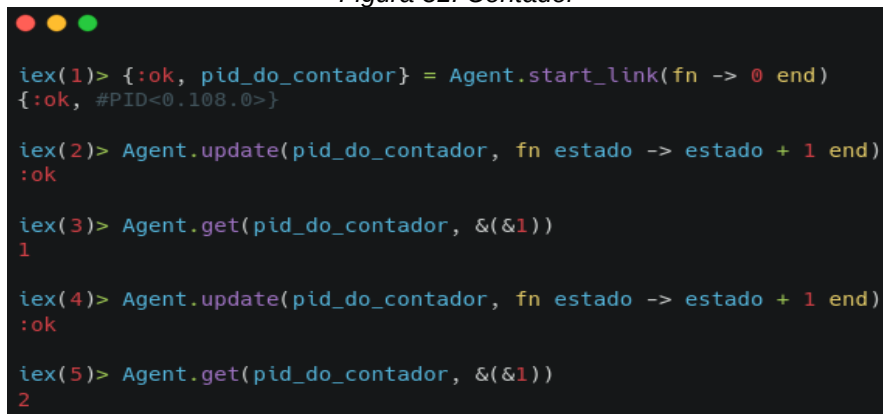
No primeiro caso, o terminal do IEx fica travado até a função `:timer.sleep()` terminar. Já no segundo caso, a execução do código não fica bloqueada em função de sua realização em segundo plano, assincronamente. O poder dessas abstrações

em Elixir, são particularmente muito úteis ao ter de lidar com operações custosas como essas.

### 5.3. CONTADOR

Uma outra possibilidade que Elixir apresenta, dá-se pelo manutenção do estado. Nos exemplos anteriores, em nenhum momento o estado do processo era armazenado, mas com Agent isso se torna possível. O que permite ser acessado por diferentes processos e em diferentes pontos no tempo. Veja a Figura 31 abaixo:

Figura 31: Contador

A terminal window with a dark background and light-colored text. It shows five interactive Elixir (iex) commands. The first command starts an Agent with a function that returns 0, returning an atom :ok and a PID. The second command updates the Agent's state to 1, returning :ok. The third command gets the state of the Agent, returning 1. The fourth command updates the state to 2, returning :ok. The fifth command gets the state, returning 2.

```
iex(1)> {:ok, pid_do_contador} = Agent.start_link(fn -> 0 end)
{:ok, #PID<0.108.0>}

iex(2)> Agent.update(pid_do_contador, fn estado -> estado + 1 end)
:ok

iex(3)> Agent.get(pid_do_contador, &(&1))
1

iex(4)> Agent.update(pid_do_contador, fn estado -> estado + 1 end)
:ok

iex(5)> Agent.get(pid_do_contador, &(&1))
2
```

Fonte: Autores

Ao dar início a um Agent, seu retorno é uma tupla com um atom :ok e o PID do processo. Isso permite que seu estado seja atualizado, com Agent.update; Ou mesmo resgatado, por meio de Agent.get.

Perceba a função start\_link, que permite vincular processos. Ao utilizá-la, o processo iniciado estará ligado e seu estado será de total conhecimento do processo pai (IEx). Isso é uma herança de Erlang, veja a seção 2.2.3. Links.

Nesse sentido, a propagação de erros pelo sistema tem sua garantia realizada. Como um processo está correlacionado com outro, pode-se construir uma estrutura pela qual o início, monitoramento e encerramento podem ser ocasionados. Isso é o princípio da árvore de supervisão, que tem em seu alicerce o conceito de supervisores (veja em 2.3.3.5. Supervisores – Supervisor).

Note ainda o uso da taquigrafia `&`. Como funções anônimas são extremamente comuns em algumas dessas situações, foi-se criada tal representação. Neste contexto, objetiva-se deixar mais enxuta a visualização deste código, veja a Tabela 10 – Operadores Importantes.

#### 5.4. PILHA

Para encerrar os exemplos, faz-se importante colocar mais uma abstração de Elixir: `GenServer`. O `Agent`, módulo aplicado acima, constitui uma parte daquele que será apresentado nesta seção. Então, pode-se imaginar a imensidão aqui tratada.

A pensar em suas bases, `GenServer` está alastrado aos comportamentos OTP de Erlang (veja 2.4. ERLANG/OTP). Em sua construção, estão implementados uma miríade de chamadas de sistemas (callbacks). De maneira mais básica, pode-se colocá-lo como um processo que reverbera uma mensagem por interação, enviando um estado atualizado.

Figura 32: Pilha

```
defmodule Stack do
  use GenServer

  # Server (callbacks)

  @impl true
  def init(stack) do
    {:ok, stack}
  end

  @impl true
  def handle_call(:pop, _from, [head | tail]) do
    {:reply, head, tail}
  end

  @impl true
  def handle_cast({:push, element}, state) do
    {:noreply, [element | state]}
  end

  # Client

  def start_link(default) when is_list(default) do
    GenServer.start_link(__MODULE__, default)
  end

  def pop(pid) do
    GenServer.call(pid, :pop)
  end

  def push(pid, element) do
    GenServer.cast(pid, {:push, element})
  end
end
```

Fonte: adaptado de HexDocs

A Figura 32 traz a construção de uma pilha através do uso de GenServer. Abaixo do nome do módulo, perceba a macro `use` (veja 4.10.6 Use) seguida do nome `GenServer`. Os comportamentos terão sua internalização no módulo `Stack`.

Outra coisa importante a ressaltar é que existem dois contextos, servidor e cliente. O primeiro, composto por funções que chamam as callbacks do GenServer. O segundo, por sua vez, compreende a API do cliente.

Antes de desenvolver sobre as funções, note o atributo `@impl` acima de cada uma. Essa anotação permite ao compilador Elixir (veja 4.5. Elixirc), avisar sobre a existência da implementação de algum comportamento OTP. Então, quem desenvolve e atenta-se para sua existência, compreenderá que são implementações de callbacks.

Por dentro do contexto do servidor, cada comportamento buscado é delimitado por uma função. Ao iniciar uma pilha, os argumentos são passados ao `init()` para que haja a configuração do estado inicial, realizada através de seu valor de retorno.

Já em `handle_call()`, tem-se o comportamento síncrono definido. Essa função recebe a requisição (`:pop`), o PID do processo que invocou (`_from`) e um estado existente (`[head | tail]`), com sua resposta esperada sendo uma tupla que contém os seguintes termos: `:reply`, a resposta (`head`) e o estado (`tail`). Como é síncrona, uma resposta é esperada para que o chamador continue seu fluxo.

Por sua vez, `handle_cast()` reforça o lado assíncrono da força. Muito similar a `handle_call()`, entretanto, não recebe o PID do processo chamador. Além disso, seu retorno não é esperado (embora retorne um `:ok`), até por conta disso o termo `:noreply` inicia a tupla internalizada. O outro elemento desta tupla, caracteriza-se por ser uma lista na qual o elemento recebido é colocado no início, o que forma o topo da Stack. Em função de ser assíncrona, a execução é não bloqueante.

No contexto do cliente, a função `start_link()` permite vincular processos. Ela recebe o nome do módulo, representado por `__MODULE__`, além de uma lista. Note a cláusula de guarda `when`, realizando a verificação do item enviado. Exigindo que o argumento inicial seja uma lista.

As funções `pop()` e `push()`, seguem um comportamento similar ao de `handle_call()` e `handle_cast()`, respectivamente: constituem abstrações para facilitar a `Stack`. No entanto, enviando solicitações a partir do lado cliente. Isso é o poder de construir em Elixir, usando comportamentos OTP.

Acima, um exemplo de pilha foi desenvolvido. Contudo, sua aplicação não foi demonstrada. Então, a Figura 33 apresenta a execução do módulo `Stack`:

*Figura 33: Pilha em execução*

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays five lines of Elixir code and their corresponding outputs. The code uses `Stack.start_link()`, `Stack.push()`, and `Stack.pop()` functions. The outputs show the process ID and the values being pushed and popped from the stack.

```
iex(1)> {:ok, pid_da_pilha} = Stack.start_link([])
{:ok, #PID<0.149.0>}

iex(2)> Stack.push(pid_da_pilha, 1)
:ok

iex(3)> Stack.push(pid_da_pilha, 2)
:ok

iex(4)> Stack.pop(pid_da_pilha)
2

iex(5)> Stack.pop(pid_da_pilha)
1
```

*Fonte: Autores*

## 6. CONCLUSÃO

A linguagem Elixir, constitui uma prolífica ferramenta para tecer códigos concorrentes. Dentro de contextos nos quais a sincronização e a comunicação são o foco, suas abstrações logram ainda mais o poderoso ecossistema sobre a qual é construída. O que garante a robustez necessária ao processo de aprendizagem.

Não fosse suficiente, Elixir traz ferramentas próprias para o ambiente. O IEx, um shell interativo (interface entre núcleo do sistema e usuário) pelo qual aplicações concorrentes ou não podem ser testadas. Ainda existe o Mix, uma ferramenta para construir e gerenciar códigos. Essas são marcas congruentes, de uma linguagem que objetiva viabilizar o ato de ensinar.

Além disso, as funções contribuem para uma parte fundamental da aprendizagem. Por dentro da praxes funcional, permite-se baixa preocupação com memória e um foco maior no algoritmo. Dessa forma, a natureza das aplicações concorrentes pode ser alcançada com menos intempestividade.

Assim, Elixir demonstra toda a proeminência para lidar com concorrência. A linguagem permite um fluxo acessível e suave para a apresentação desta forma de programar. Uma ótima porta de entrada ao processo de aprendizagem.

### 6.1. RECOMENDAÇÕES DE ESTUDOS FUTUROS

Para continuar os estudos com esta linguagem, sugere-se lidar com a programação distribuída. O modo como Elixir lida com o tratamento de erros (veja 2.2.2. Tratamento de Erros) e quais abstrações de comportamentos são incorporados para quem desenvolve com esta ferramenta. Por exemplo, a coordenação entre nós nomeados (veja 2.3.1. Nó Nomeado) para a realização de tarefas distribuídas.



## REFERÊNCIAS

ALMEIDA, Ulisses. **Learn Functional Programming with Elixir**. Pragmatic Bookshelf, 2018.

ARMSTRONG, Joe. A history of Erlang. In: **Proceedings of the third ACM SIGPLAN conference on History of programming languages**. 2007. p. 6-1-6-26.

ARMSTRONG, Joe. **Programming Erlang: software for a concurrent world**. Pragmatic Bookshelf, 2013.

BEN-ARI, Mordechai. **Principles of concurrent programming**. Prentice Hall Professional Technical Reference, 1982.

BRANAS, Rodrigo. **Elixir com José Valim // Live #61**. Entrevistador: Rodrigo Branas. Youtube, [s. d.]. 1 vídeo (86,03 min). [S, l.: s. n.]. Disponível em: <<https://www.youtube.com/watch?v=EXE7NUxBKrk>>. Acesso em: 08 Out. 2021.

CONFERÊNCIA APA. **Desenvolvendo sistemas com Elixir – José Valim**. Apresentador: Filipe Mulonde. Youtube, 20 nov. 2020. 1 vídeo (90,23 min). [S, l.: s. n.]. Disponível em: <[https://www.youtube.com/watch?v=mV\\_C5wplowg](https://www.youtube.com/watch?v=mV_C5wplowg)>. Acesso em: 08 Out. 2021.

DAVI, Tiago. **Elixir: Do zero à concorrência**. Editora Casa do Código, 2017.

GUSTAVSSON, Björn. A Brief History of The BEAM Compiler. In: **A Blog from the Erlang/OTP team**. 2018. Disponível em: <https://blog.erlang.org/beam-compiler-history/>. Acessado em: 01 Out. 2021.

HACKERS / FOUNDERS. **An Evening at Erlang Factory: Joe Armstrong, Mike Williams, Robert Virding**. Youtube, 29 Abr. 2015. 1 vídeo (35,50 min). [S, l.: s. n.]. Disponível em: <[https://www.youtube.com/watch?v=rYkIO\\_ixRDc](https://www.youtube.com/watch?v=rYkIO_ixRDc)>. Acesso em: 09 Out. 2021.

HEBERT, Fred. **Learn you some Erlang for great good!: A Beginner's Guide**. No Starch Press, 2013. Disponível em: <<https://learnyousomeerlang.com/content>>. Acesso em: 09 Out. 2021.

HÖGBERG, John. A brief introduction to BEAM. In: **A Blog from the Erlang/OTP team**. 2020. Disponível em: <https://blog.erlang.org/a-brief-BEAM-primer/>. Acessado em: 02 Out. 2021.

LAURENT, Simon St; EISENBERG, J. David. **Introducing Elixir: Getting Started in Functional Programming**. "O'Reilly Media, Inc.", 2016.

LOGAN, Martin; MERRITT, Eric; CARLSSON, Richard. **Erlang and OTP in Action**. Manning Publications Co., 2010.

MANZANO, José Augusto NG. **Introdução a programação funcional (8)**. Youtube, 17 set. 2021. 1 vídeo (83,03 min). São Paulo. [s. n.]. Disponível em: <<https://www.youtube.com/watch?v=FZ-FnGfuipM>>. Acesso em: 07 Out. 2021.

ROCKETSEAT. **Especial: entrevista exclusiva com José Valim, criador da linguagem Elixir | Podcast Faladev #26**. Entrevistador: Diego Fernandes. Youtube, 20 Nov. 2020. 1 vídeo (73,266667 min). [S, l.: s. n.]. Disponível em: <<https://www.youtube.com/watch?v=roc-5euwaY>>. Acesso em: 08 Out. 2021.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação-9**. Bookman Editora, 2011.

STENMAN, Erik. **The Erlang Runtime System**. GitHub, 2020. Disponível em: <https://github.com/happi/theBeamBook>. Acessado em: 02 Out. 2021.

STACKOVERFLOW. Programação Concorrente x Paralela x Distribuída. In: **Stack Overflow**. 2015. Disponível em: <https://pt.stackoverflow.com/questions/75727/programação-concorrente-x-paralela-x-distribuída>. Acessado em: 04 Out. 2021.

WATANABE, Elaine Naomi. Além da Programação Funcional com Elixir e Erlang. In: **Speakerdeck**. 2019. Disponível em: <https://speakerdeck.com/elainenaomi/alem-da-programacao-funcional-com-elixir-e-erlang>. Acessado em: 02 Out. 2021.

\_\_\_\_\_. Erlang Reference Manual User's Guide. In: **Erlang.org**. 2021. Disponível em: [https://erlang.org/doc/reference\\_manual/users\\_guide.html](https://erlang.org/doc/reference_manual/users_guide.html). Acessado em: 04 Out. 2021.

\_\_\_\_\_. Erlang Run-Time System Application (ERTS) Reference Manual. In: **Erlang.org**. 2021. Disponível em: <https://erlang.org/doc/apps/erts/>. Acessado em: 04 Out. 2021.

\_\_\_\_\_. OTP Design Principles User's Guide. In: **Erlang.org**. 2021. Disponível em: [https://erlang.org/documentation/doc-12.1/doc/design\\_principles/users\\_guide.html](https://erlang.org/documentation/doc-12.1/doc/design_principles/users_guide.html). Acessado em: 04 Out. 2021.

\_\_\_\_\_. STDLIB-3.16. In: **Erlang.org**. 2021. Disponível em: <https://erlang.org/documentation/doc-12.1/lib/stdlib-3.16/doc/html/supervisor.html>. Acessado em: 04 Out. 2021.

\_\_\_\_\_. Elixir. In: **Elixir-lang**. 2021. Disponível em: <https://elixir-lang.org/>. Acessado em: 14 Out. 2021.

\_\_\_\_\_. HexDocs. In: **HexDocs**. 2021. Disponível em: <https://hexdocs.pm/>. Acessado em: 14 Out. 2021.

\_\_\_\_\_. Elixir School. In: **ElixirSchool**. 2021. Disponível em: <https://elixirschool.com/>. Acessado em: 14 Out. 2021.

\_\_\_\_\_. O Sistema Operacional GNU. In: **gnu.org**. 2021. Disponível em: <https://www.gnu.org/licenses/license-list.html>. Acessado em: 14 Out. 2021.