Introdução à Programação Funcional com Haskell

Henrique de Souza Santana

henrique.s.santana@ufv.br

Antes de começar...

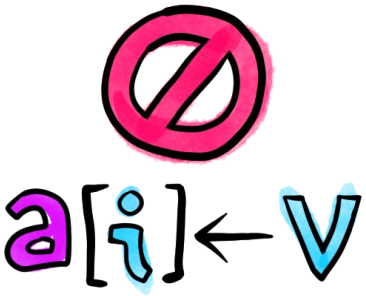
Ambientação

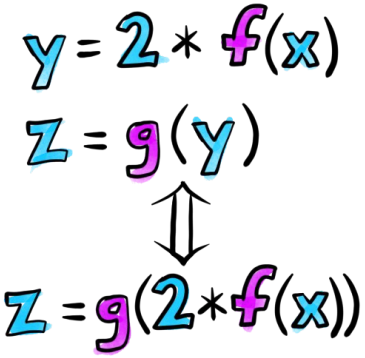
- Verifique se já possui o compilador e interpretador da linguagem.
  - Abra o terminal e execute `ghci`.
- Se precisar instalar, vá para a página do GHCup (haskell.org/ghcup) e execute o comando indicado para seu sistema operacional.
  - Escolha sim (Y) para todas as opções de instalação.
  - Vídeo mostrando a instalação no Windows: youtu.be/bB4fmQiUYPw

O que é programação funcional?

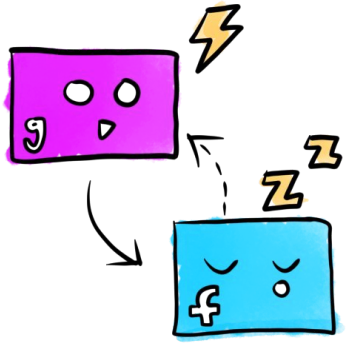
O que é programação funcional?

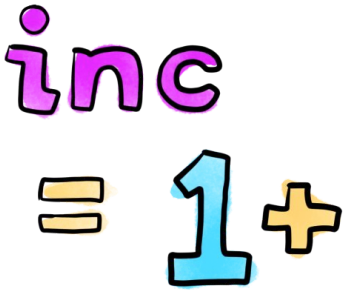
- Paradigma baseado na aplicação e composição de funções;
- Imutabilidade;
- Ausência de efeitos colaterais;
- Transparência referencial;





O que é programação funcional?





- Funções de ordem superior;
- Avaliação preguiçosa;
- Aplicação parcial;

História do paradigma

Cálculo Lambda por
Alonzo Church

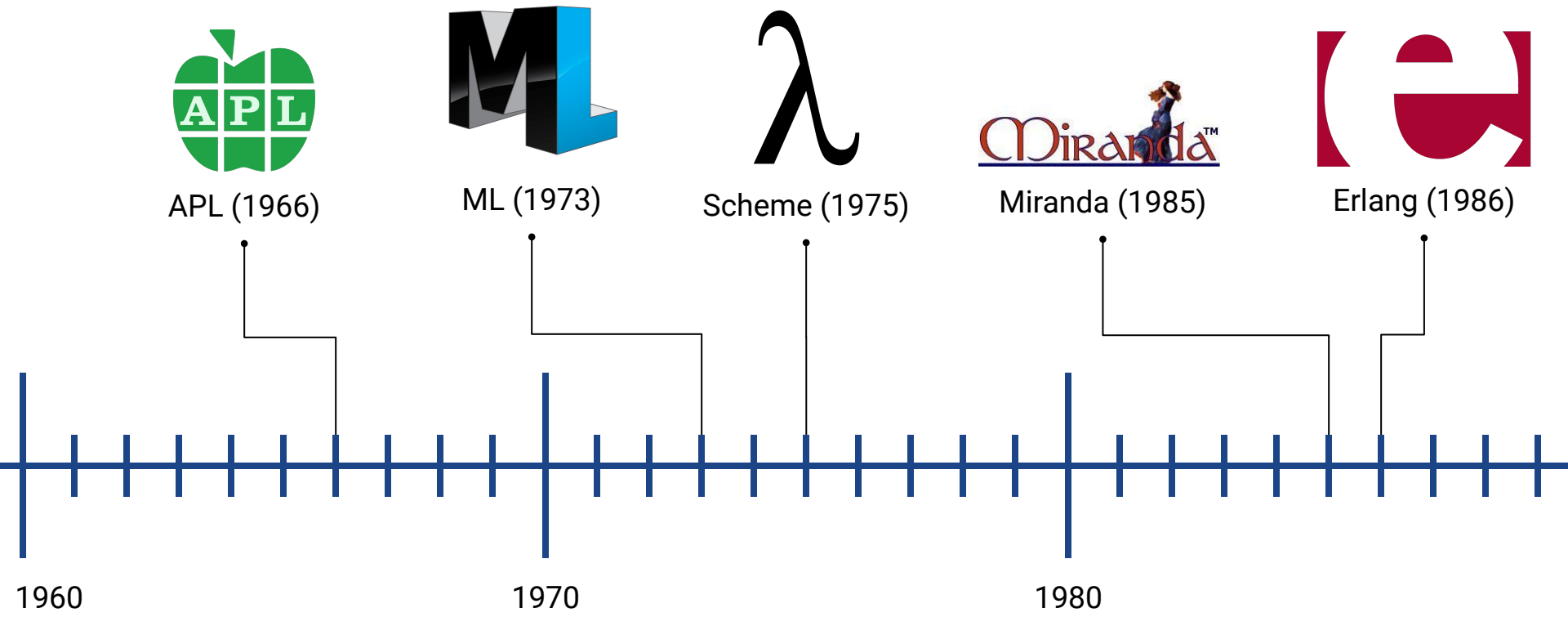
Lisp (1958)

1930

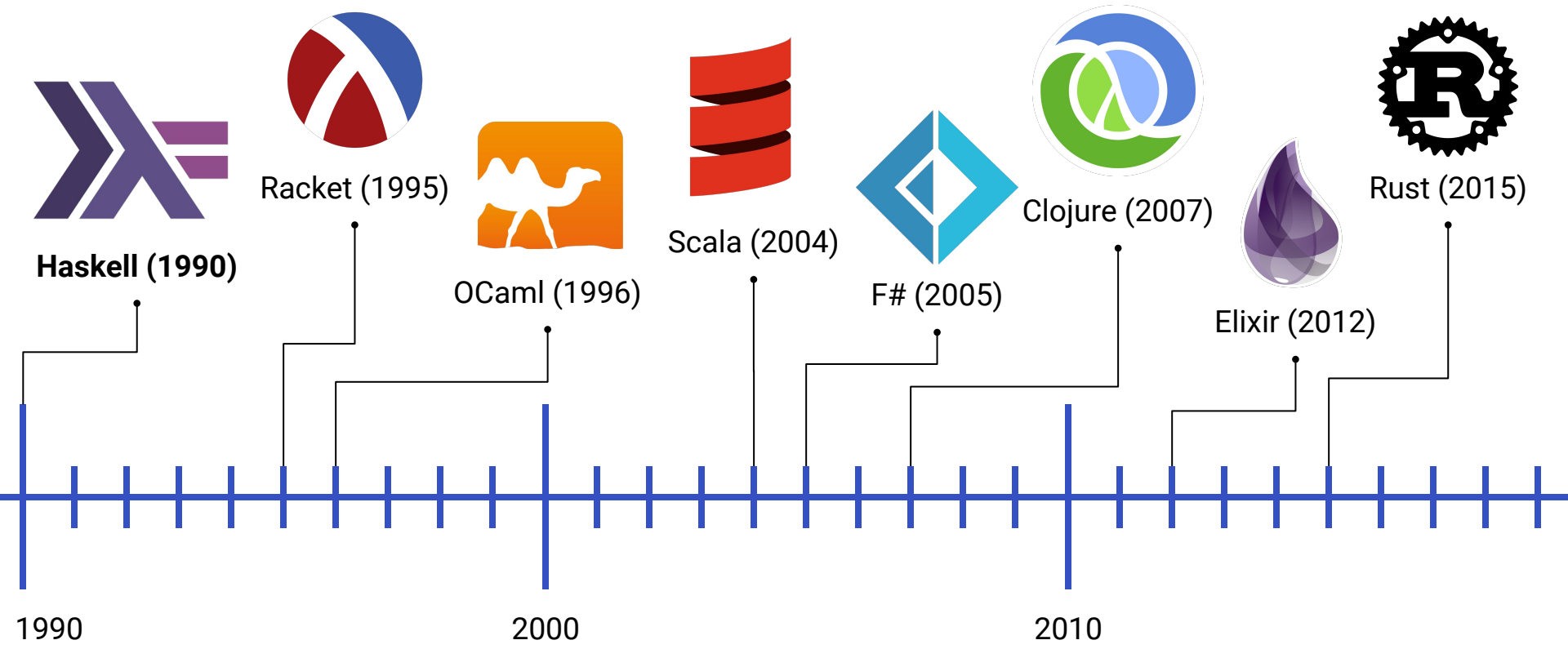
1940

1950

História do paradigma



História do paradigma



Praticando com Haskell

Hello, World!

- Opção 1: executar diretamente pelo `ghci`

```
ghci> putStrLn "Hello, World!"
```

- Opção 2: salvar o código no arquivo `hello.hs`, abrir o `ghci`, carregar o arquivo com `:l hello.hs`, e chamar a função `hello`.

```
hello = putStrLn "Hello, World!"
```

Conhecendo a sintaxe

Operações básicas:

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
```

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
```

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

Conhecendo a sintaxe

Funções:

```
dobro x = x * 2
-- ou então
dobro = (* 2)

ghci> dobro 4
8
ghci> dobro 8 + dobro 13
42
```

```
delta a b c = b**2 - 4*a*c

ghci> delta 2.5 4.0 1.0
6.0

ghci> delta (-1) 2 3
16.0
```

Exercício - Bhaskara

- Vamos implementar uma função que consiga informar as soluções de uma equação de segundo grau:

$$ax^2 + bx + c = 0$$

$$\Delta = b^2 - 4ac$$

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

Conhecendo a sintaxe

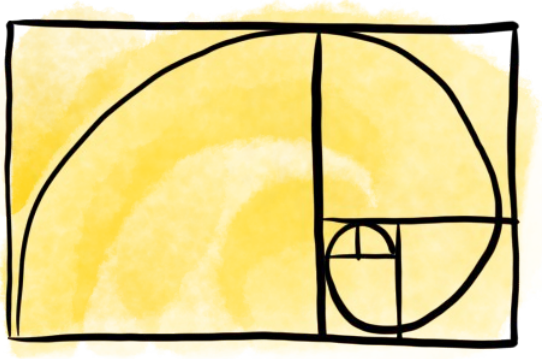
Funções:

```
fatorial :: Int -> Int
fatorial 0 = 1
fatorial n = n * factorial (n - 1)

ghci> fatorial 5
120
```

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

ghci> fib 7
13
```

Exercício - Melhorando Fibonacci

- Como melhorar essa função Fibonacci?



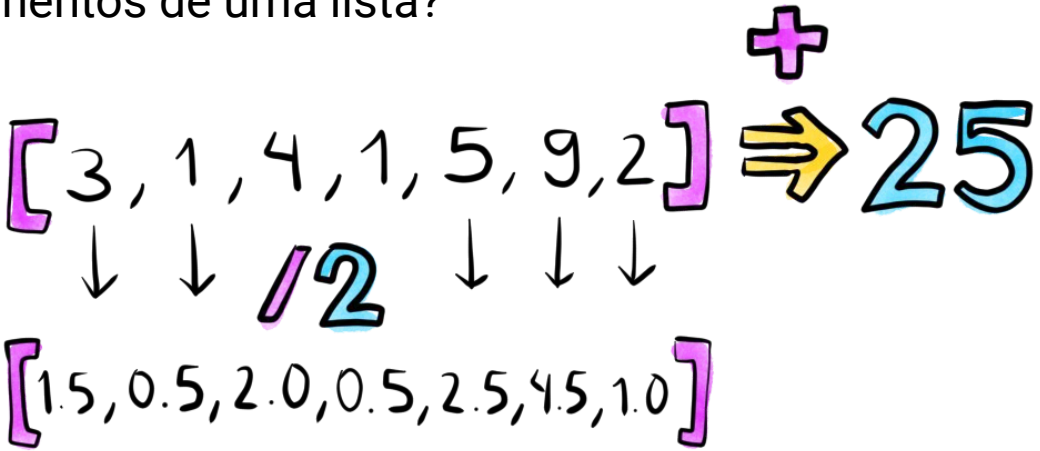
Listas

```
ghci> [0,1] ++ [1,2,3,5]
[0,1,1,2,3,5]

-- String é o mesmo que [Char]
ghci> 'A':"BCD"
"ABCD"

ghci> [0,2..10] !! 3
6

ghci> [0,2..] !! 3
6
```

- Toda lista é construída e desconstruída com (`:`)
- Várias funções prontas:
  - `length`
  - `reverse`
  - `head`
  - `tail`
  - `take`
  - `drop`
  - `sum`
  - `product`
  - `maximum`
  - `minimum`

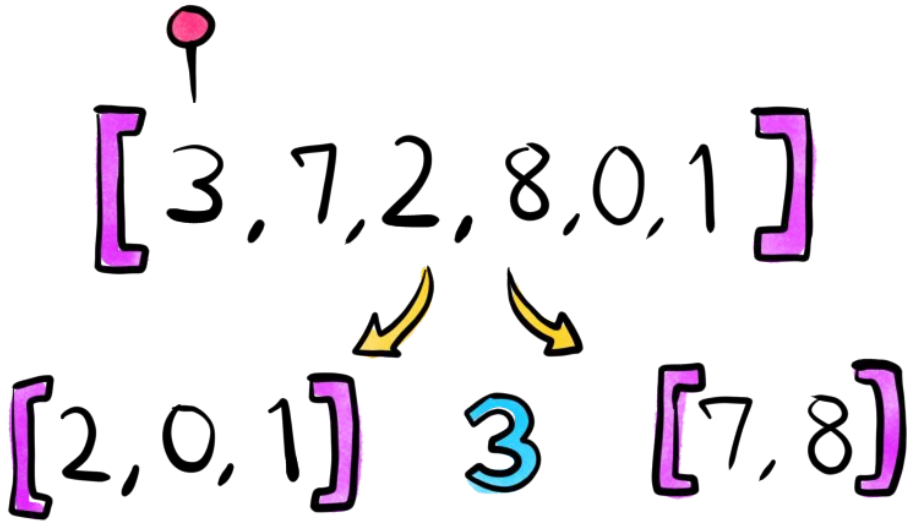
Exercício - Funções de Ordem superior

- Como percorrer uma lista?
- Como somar os elementos de uma lista?
- Como filtrar os elementos de uma lista?
- Como transformar os elementos de uma lista?



Exercício - Quicksort

- Como implementar (uma versão do) Quicksort em Haskell?



Tuplas

```
ghci> :t ('A',"abacate")
('A',"abacate") :: (Char, String)

ghci> zip [1..] ['A'..'D']
[(1,'A'),(2,'B'),(3,'C'),(4,'D')]

ghci> fst (2.0, -5.7)
2.0

ghci> snd (2.0, -5.7)
-5.7
```

- Coleção heterogênea de dados, mas de tamanho fixo;
- Geralmente usamos tuplas de tamanho 2;
- Funções prontas:
  - `fst`
  - `snd`

Maybe

```
ghci> :t Nothing
Nothing :: Maybe a

ghci> :t Just "Oi!"
Just "Oi!" :: Maybe String

ghci> lookup 20 (zip [1..] ['A'..'Z'])
Just 'T'

ghci> lookup 27 (zip [1..] ['A'..'Z'])
Nothing
```

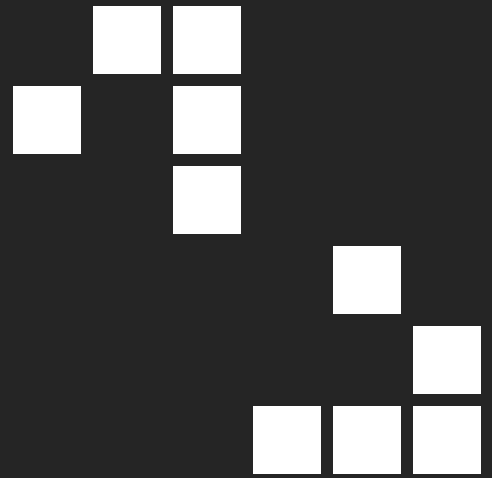
- Alternativa aos valores nulos;
- *Talvez* contém um valor;
- Funções prontas:
  - `fromMaybe`
  - `fromJust`
  - `isNothing`
  - `isJust`
  - `catMaybes`
  - `listToMaybe`
  - `maybeToList`

Exercício - Pesquisa

- Como implementar uma pesquisa sequencial, usando um critério de busca arbitrário?

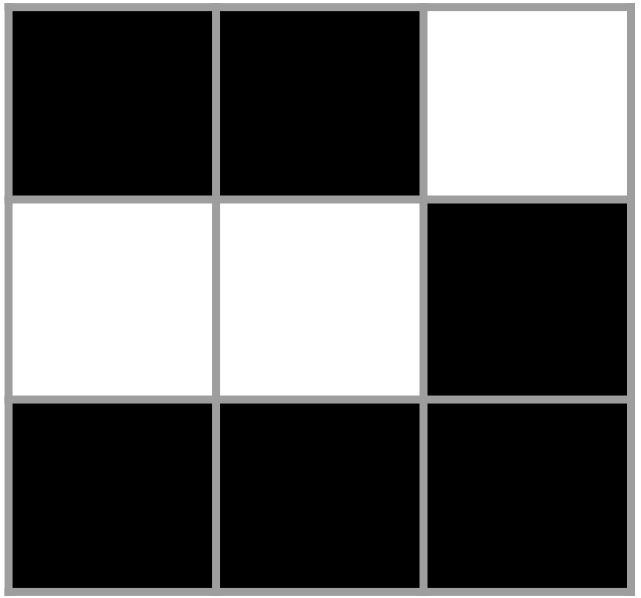
```
lookupBy :: (a -> Bool) -> [a] -> Maybe a
lookupBy f xs = ???
```

Jogo da Vida



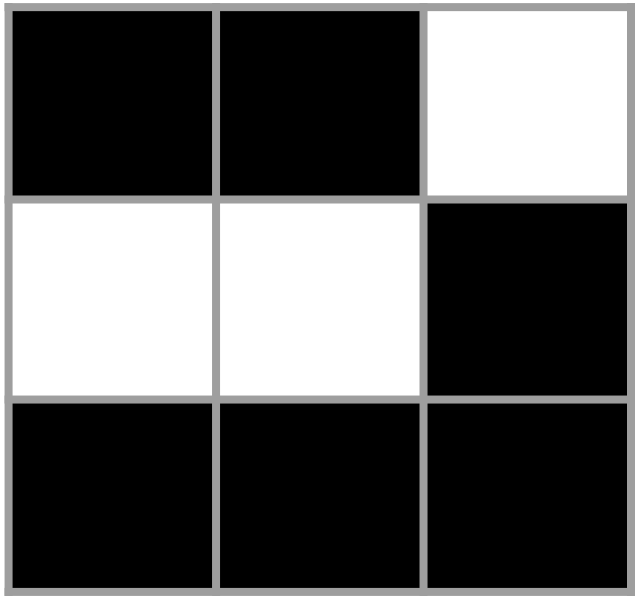
Jogo da Vida de Conway

- Autômato celular;
- Tabuleiro bidimensional contendo células vivas ou mortas;
- Jogo de zero-jogador;
- Apenas o estado inicial é dado;
- O jogo evolui de geração em geração seguindo quatro regras.



Jogo da Vida de Conway

1. Toda célula morta com exatamente três vizinhos vivos torna-se viva por nascimento.
2. Toda célula viva com menos de dois vizinhos vivos morre por isolamento.
3. Toda célula viva com mais de três vizinhos vivos morre por superpopulação.
4. Toda célula viva com dois ou três vizinhos vivos permanece viva.



Dúvidas?

Obrigado pela atenção!

Henrique de Souza Santana

henrique.s.santana@ufv.br

Referências

- LIPOVACA, Miran. **Learn you a haskell for great good!: a beginner's guide**. no starch press, 2011. Disponível em: http://learnyouahaskell.com/
- Why Functional Programming Matters. [Locução de]: Eric Normand. LispCast: 12 jul. 2020. *Podcast*. Disponível em: https://ericnormand.me/podcast/why-functional-programming-matters
- HUGHES, John. Why Functional Programming Matters. **The computer journal**, v. 32, n. 2, p. 98-107, 1989. Disponível em: https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf