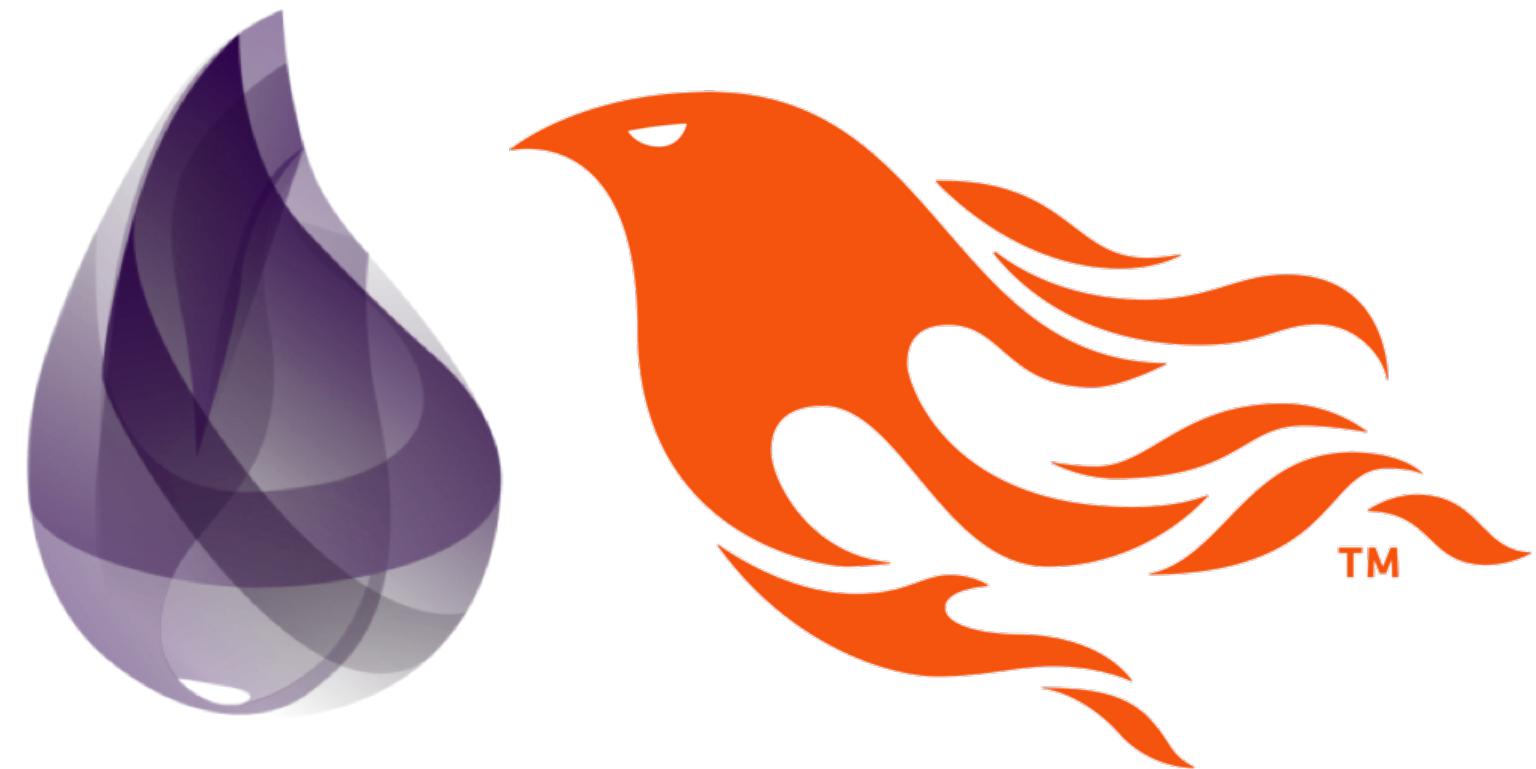


Desenvolvimento web com
Elixir e Phoenix





Andrés Vidal

Técnico em Informática



Engenheiro da Computação



Engenheiro de Software
WYEWORKS.

<https://github.com/andres-vidal/elixir-course>

Paradigma de
Programação Funcional

Funções como cidadãos de primeira classe

Funções como cidadãos de primeira classe

**Podem ser asignadas a
variáveis**

Funções como cidadãos de primeira classe

Podem ser asignadas a variáveis

Podem ser passadas como argumentos para outras funções

Funções como cidadãos de primeira classe

Podem ser asignadas a variáveis

Podem ser passadas como argumentos para outras funções

Podem ser retornadas desde outras funções

Funções como cidadãos de primeira classe

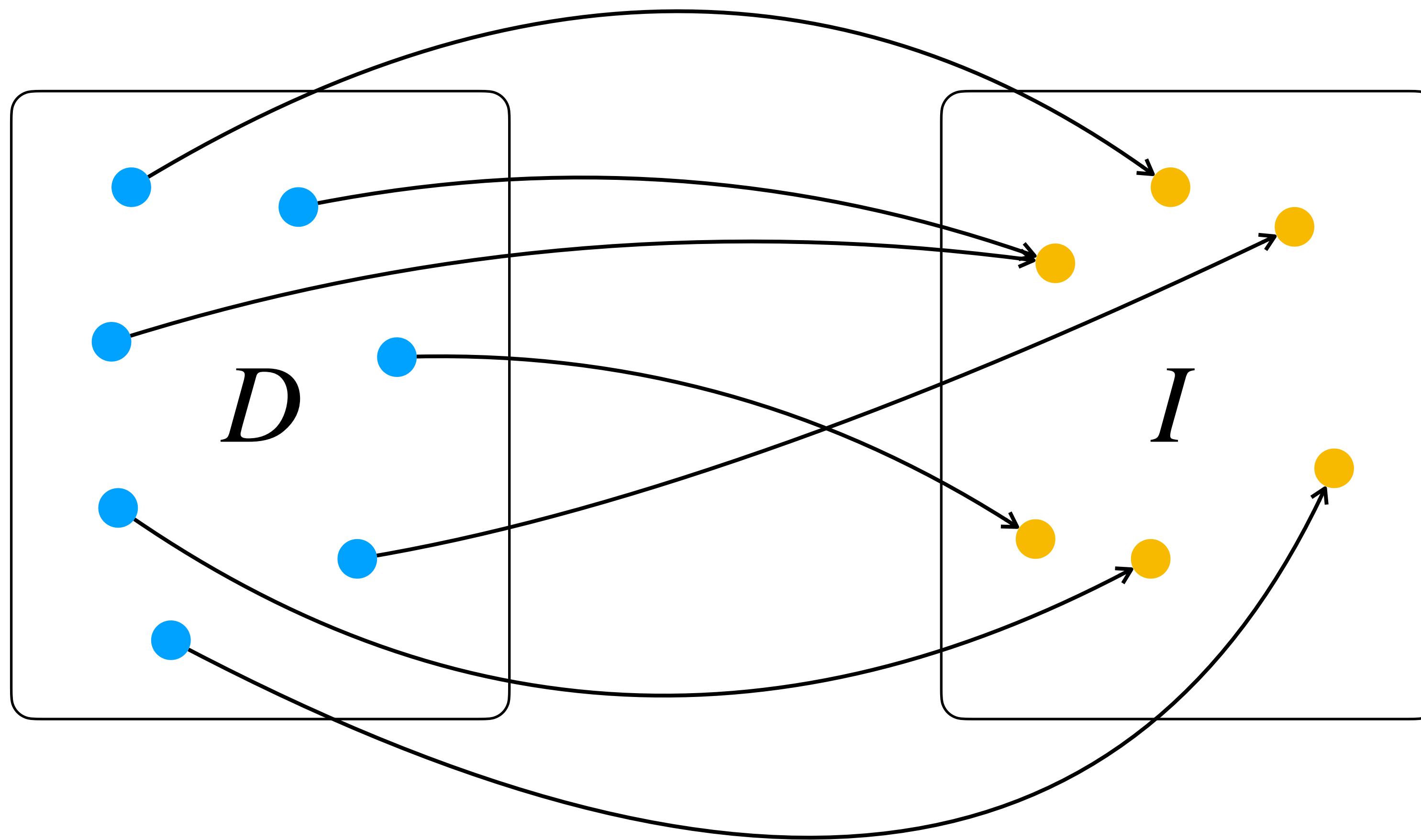
Podem ser asignadas a variáveis

Podem ser passadas como argumentos para outras funções

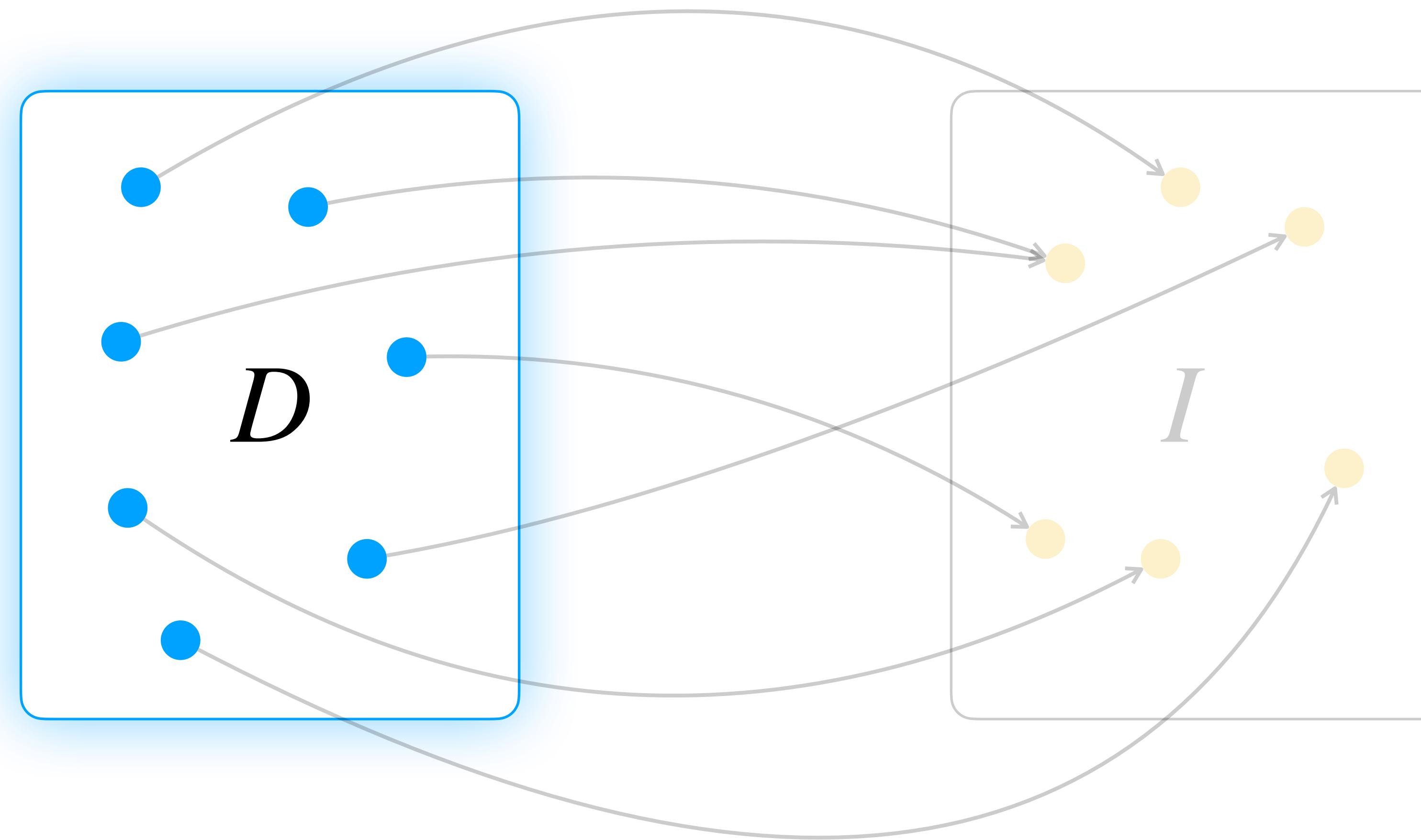
Podem ser retornadas desde outras funções

Isto é, podem ser tratadas como qualquer valor
(números, strings, vetores, structs)

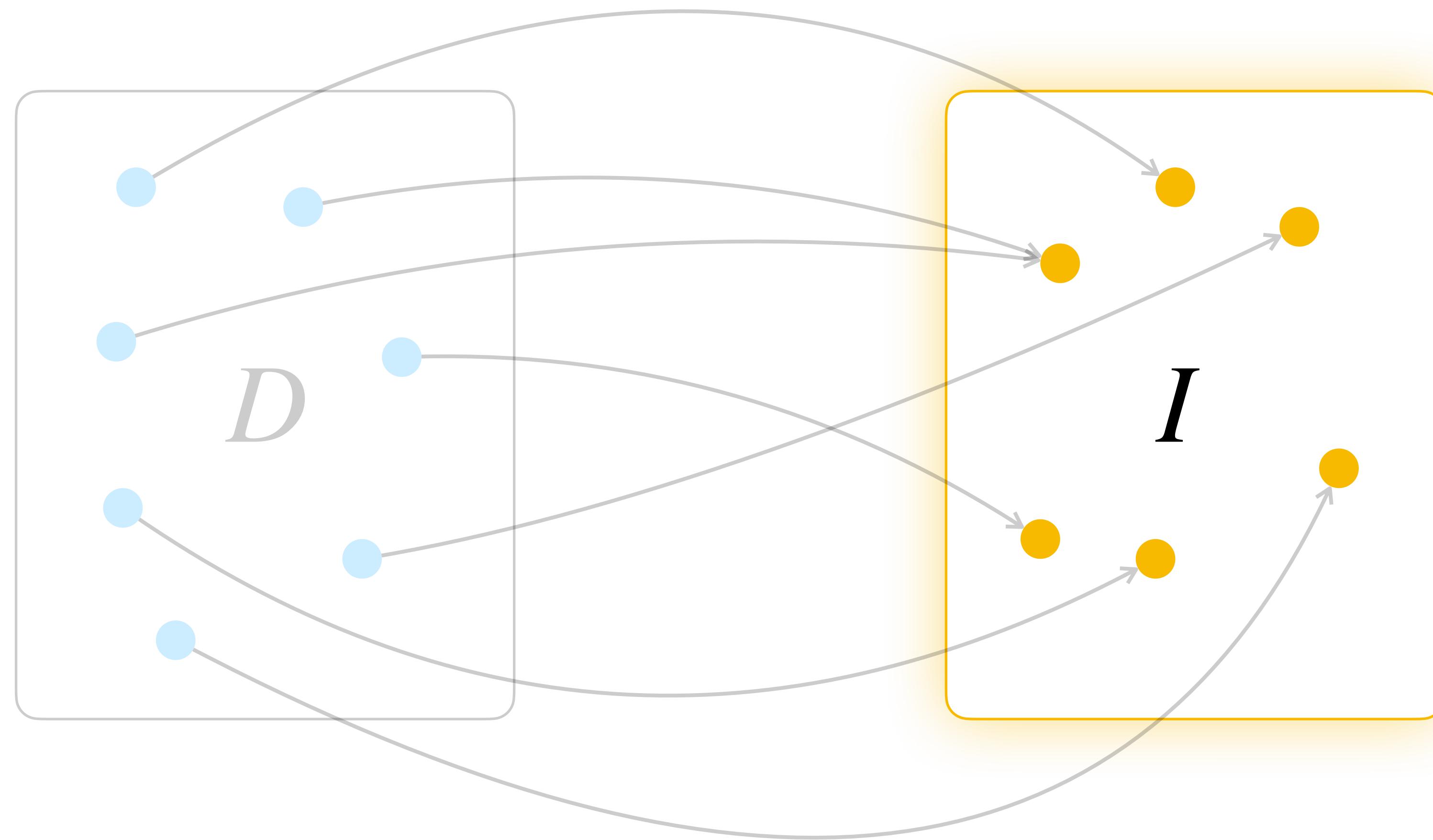
$$f: D \rightarrow I$$



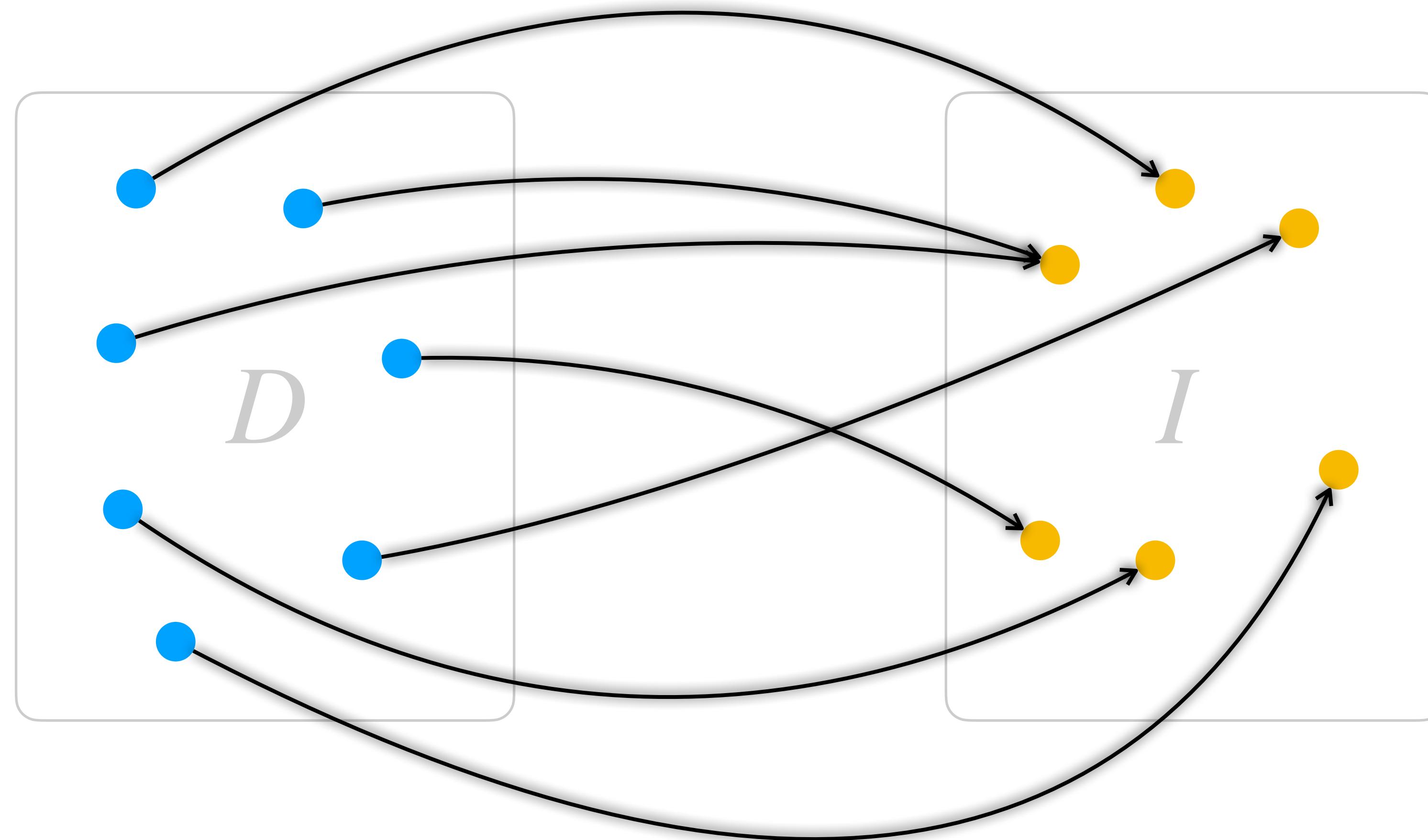
$$f: D \rightarrow I$$



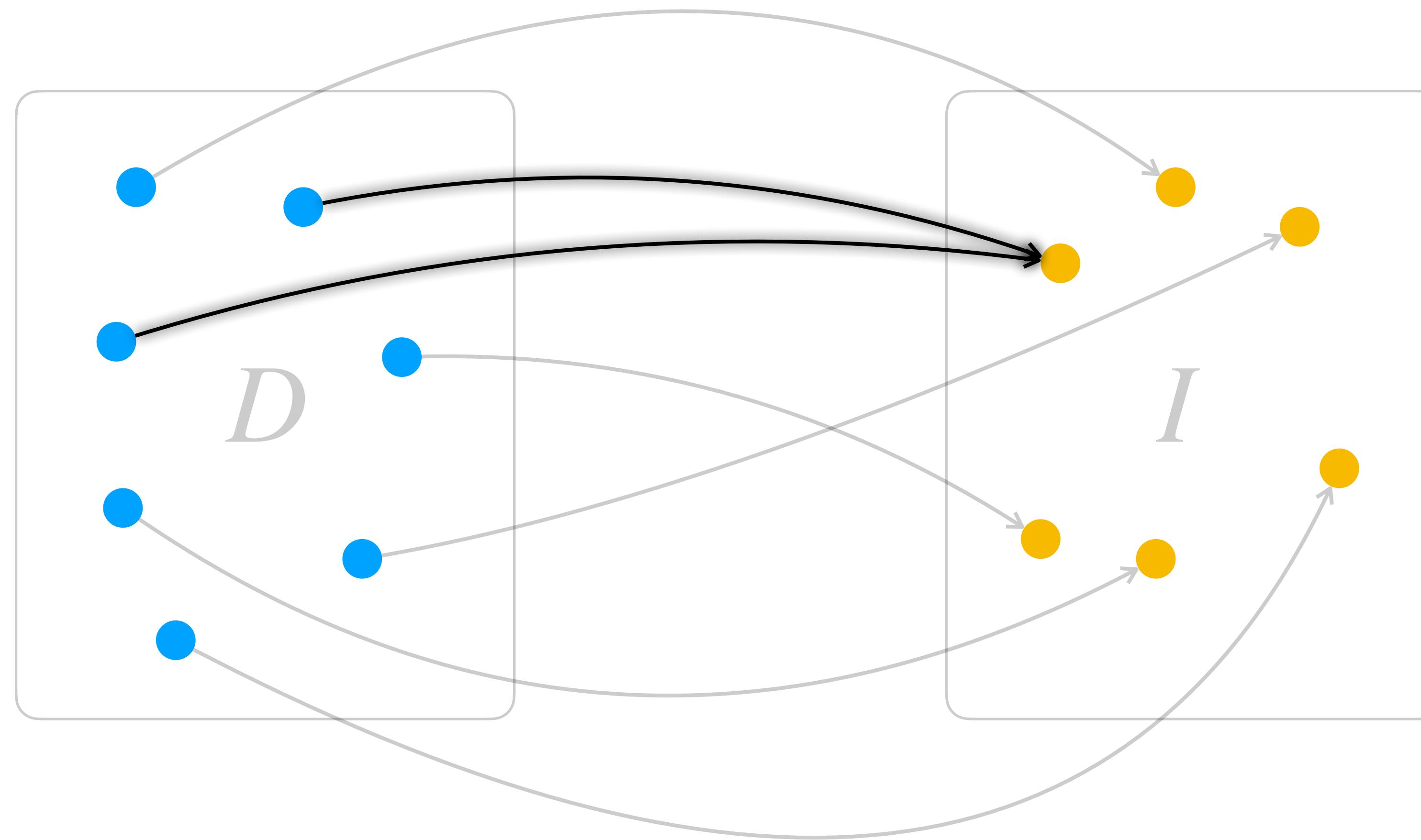
$$f: D \rightarrow I$$



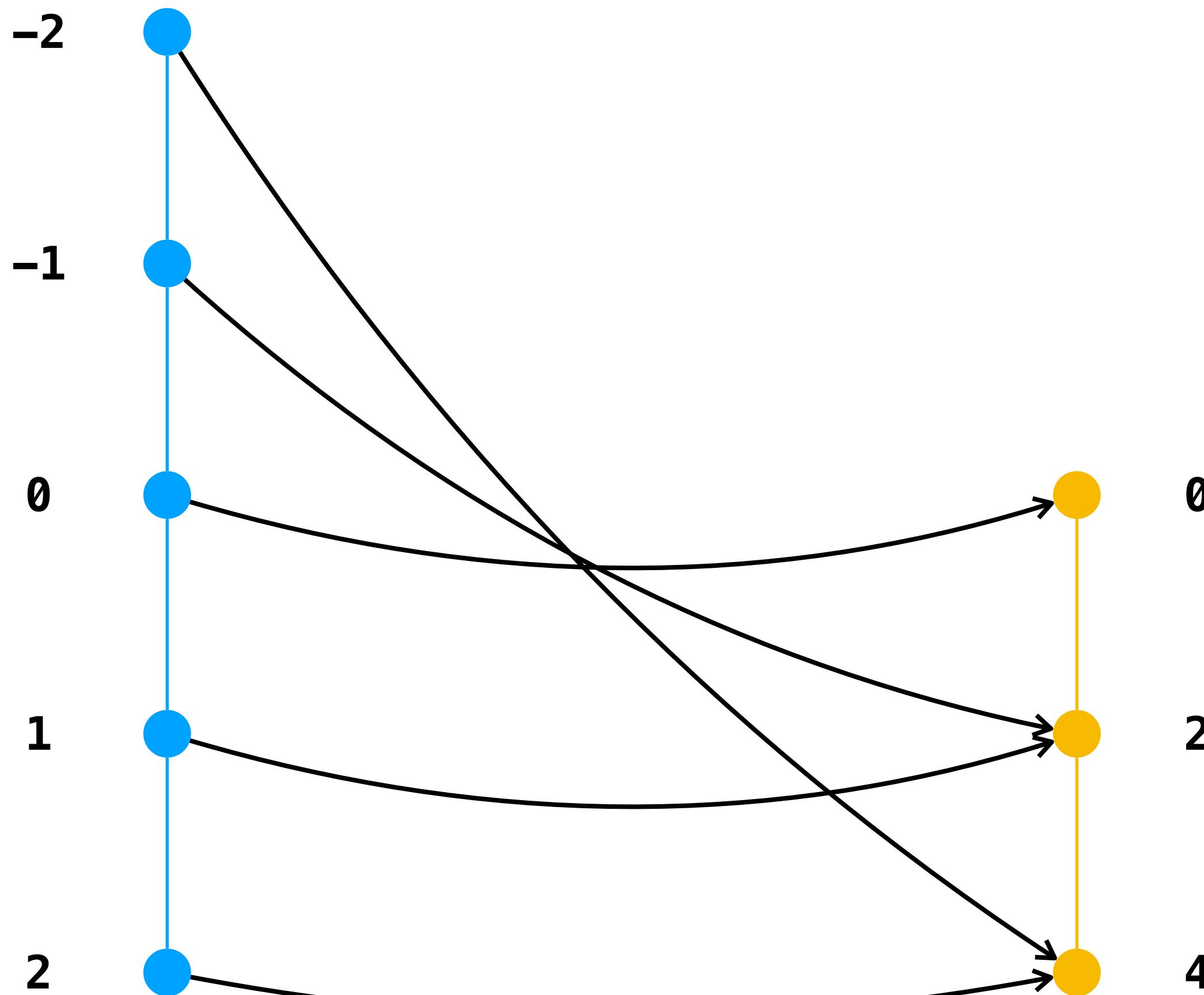
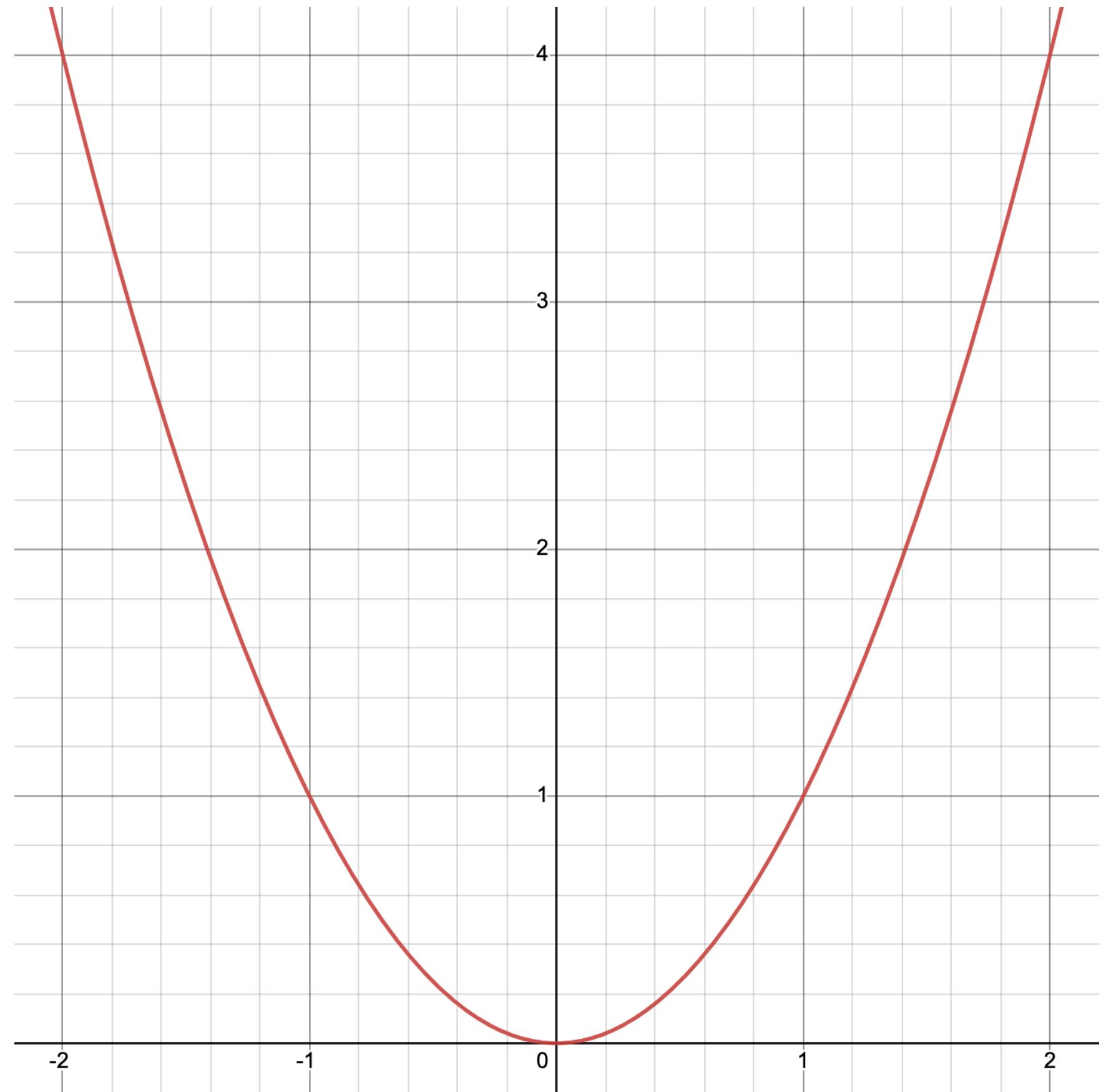
$$f: D \rightarrow I$$



$$f: D \rightarrow I$$



$$f: \mathbb{R} \rightarrow \mathbb{R}^+ \quad | \quad f(x) = x^2$$



Funções podem ser **compostas**

$$f(x) = x^2 + h(x) - g(x) \times z(x)$$

Funções podem ser **compostas**

$$f(x) = x^2 + h(x) - g(x) \times z(x)$$

podem receber **varias variáveis**

$$f(x, y) = x + y$$

Funções podem ser **compostas**

$$f(x) = x^2 + h(x) - g(x) \times z(x)$$

podem receber **varias variáveis**

$$f(x, y) = x + y$$

e podem ser **condicionais**

$$\max(x, y) = \begin{cases} x & \text{se } x > y \\ y & \text{se não} \end{cases}$$

Na matemática também temos estruturas de dados

Na matemática também temos estruturas de dados

vetores $[1, 2, 3, 4, 5] \in \mathbb{N}^5$

Na matemática também temos estruturas de dados

vetores $[1, 2, 3, 4, 5] \in \mathbb{N}^5$

matrizes $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in M_{2 \times 3}(\mathbb{N})$

Na matemática também temos estruturas de dados

vetores $[1, 2, 3, 4, 5] \in \mathbb{N}^5$

matrizes $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in M_{2 \times 3}(\mathbb{N})$

conjuntos $\{1, 2, 3, 4, 5\} \subset \mathbb{N}$

E maneiras de transformar sobre essas estruturas de dados

$$f([v_1, v_2, v_3]) = [h(v_1), g(v_2), z(v_3)]$$

E maneiras de transformar sobre essas estruturas de dados

$$f([v_1, v_2, v_3]) = [h(v_1), g(v_2), z(v_3)]$$

...com funções

Como programaríamos esta função?

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

Como programaríamos esta função?

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

```
def f(v) do
  res = new Array(length(v))

  for i in 1..length(v) do
    res[i] = h(v[i])
  end

  return res
end
```

Como programaríamos esta função?

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

```
def f(v) do
  res = new Array(length(v))

  for i in 1..length(v) do
    res[i] = h(v[i])
  end

  return res
end
```

Focamos no **procedimento**
para obter o resultado que queremos

**Estamos acostumados a processar estruturas de dados
com laços iterativos**

**Estamos acostumados a processar estruturas de dados
com laços iterativos**

for

**Estamos acostumados a processar estruturas de dados
com laços iterativos**

for foreach

**Estamos acostumados a processar estruturas de dados
com laços iterativos**

for foreach while

**Estamos acostumados a processar estruturas de dados
com laços iterativos**

for foreach while do...while

**Estamos acostumados a processar estruturas de dados
com laços **iterativos****

~~for~~

~~foreach~~

~~while~~

~~do...while~~

Na programação funcional **NÃO temos esses recursos !**

Como programaríamos esta função sem iteração?

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

Como programaríamos esta função sem iteração?

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

```
def f(v) do
  if size(v) = 0 do
    return []
  else
    first = head(v)
    rest = tail(v)
    return h(first) ++ f(rest)
  end
end
```

Como programaríamos esta função sem iteração?

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

```
def f(v) do
  if size(v) = 0 do
    return []
  else
    first = head(v)
    rest = tail(v)
    return h(first) ++ f(rest)
  end
end
```

Utilizamos **recursividade** e
focamos no **resultado** que queremos

Como podemos garantir que uma função recursiva vai se concluir?

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

$$\textbf{Pares}_{\mathbb{N}} = \{2n : n \in \mathbb{N}\}$$

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

$$\textbf{Pares}_{\mathbb{N}} = \{2n : n \in \mathbb{N}\}$$

Por compreensão

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

$$\textbf{Pares}_{\mathbb{N}} = \{2n : n \in \mathbb{N}\}$$

$$\mathbb{N} \left\{ \begin{array}{l} 1 \in \mathbb{N} \\ \text{se } n \in \mathbb{N}, \text{ então } n + 1 \in \mathbb{N} \end{array} \right.$$

Por compreensão

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

$$\textbf{Pares}_{\mathbb{N}} = \{2n : n \in \mathbb{N}\}$$

$$\mathbb{N} \left\{ \begin{array}{l} 1 \in \mathbb{N} \text{ Passo base} \\ \text{se } n \in \mathbb{N}, \text{ então } n + 1 \in \mathbb{N} \end{array} \right.$$

Por compreensão

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

$$\textbf{Pares}_{\mathbb{N}} = \{2n : n \in \mathbb{N}\}$$

$$\mathbb{N} \left\{ \begin{array}{l} 1 \in \mathbb{N} \quad \text{Passo base} \\ \text{se } n \in \mathbb{N}, \text{ então } n + 1 \in \mathbb{N} \\ \quad \quad \quad \text{Passo indutivo} \end{array} \right.$$

Por compreensão

Na matemática podemos definir conjuntos de diferentes formas

$$\mathbb{N} = \{1, 2, 3, 4, \dots\}$$

Pares_N = { $2n : n \in \mathbb{N}$ }

Por compreensão

$$\mathbb{N} \left\{ \begin{array}{l} 1 \in \mathbb{N} \text{ Passo base} \\ \text{se } n \in \mathbb{N}, \text{ então } n + 1 \in \mathbb{N} \\ \text{Passo indutivo} \end{array} \right.$$

Por indução

Para garantir que uma função recursiva vai se completar podemos utilizar o esquema de recursão primitiva

$$\mathbf{soma}(x, n) = \begin{cases} \mathbf{soma}(x, 1) = x + 1 \\ \mathbf{soma}(x, n + 1) = \mathbf{soma}(x, n) + 1 \end{cases}$$

Para garantir que uma função recursiva vai se completar podemos utilizar o **esquema de recursão primitiva**

$$\mathbf{soma}(x, n) = \begin{cases} \mathbf{soma}(x, 1) = x + 1 \\ \mathbf{soma}(x, n + 1) = \mathbf{soma}(x, n) + 1 \end{cases}$$

1. Uma definição **primitiva** por passo base

Para garantir que uma função recursiva vai se completar podemos utilizar o **esquema de recursão primitiva**

$$\mathbf{soma}(x, n) = \begin{cases} \mathbf{soma}(x, 1) = x + 1 \\ \mathbf{soma}(x, n + 1) = \mathbf{soma}(x, n) + 1 \end{cases}$$

1. Uma definição **primitiva** por passo base
2. Uma definição **recursiva** por passo **indutivo**

$v = [v_1, v_2, \dots, v_n] \in \mathbb{N}^n$

append : $\mathbb{N} \times \mathbb{N}^n \rightarrow \mathbb{N}^{n+1}$ **append**(n, v) = $[x, v_1, \dots, v_n]$

$$v = [v_1, v_2, \dots, v_n] \in \mathbb{N}^n$$

$$\mathbf{append} : \mathbb{N} \times \mathbb{N}^n \rightarrow \mathbb{N}^{n+1} \quad \mathbf{append}(n, v) = [x, v_1, \dots, v_n]$$

$$V_{\mathbb{N}} \left\{ \begin{array}{l} [] \in V_{\mathbb{N}} \\ \textbf{se } n \in \mathbb{N} \textbf{ e } v \in V_{\mathbb{N}}, \textbf{então } \mathbf{append}(n, v) \in V_{\mathbb{N}} \end{array} \right.$$

$$v = [v_1, v_2, \dots, v_n] \in \mathbb{N}^n \quad \textbf{append} : \mathbb{N} \times \mathbb{N}^n \rightarrow \mathbb{N}^{n+1} \quad \textbf{append}(n, v) = [x, v_1, \dots, v_n]$$

$$V_{\mathbb{N}} \left\{ \begin{array}{l} [] \in V_{\mathbb{N}} \\ \textbf{se } n \in \mathbb{N} \text{ e } v \in V_{\mathbb{N}}, \text{ então } \textbf{append}(n, v) \in V_{\mathbb{N}} \end{array} \right.$$

$$f: V_{\mathbb{N}} \rightarrow V_{\mathbb{N}}$$

$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

$$v = [v_1, v_2, \dots, v_n] \in \mathbb{N}^n \quad \textbf{append} : \mathbb{N} \times \mathbb{N}^n \rightarrow \mathbb{N}^{n+1} \quad \textbf{append}(n, v) = [x, v_1, \dots, v_n]$$

$$V_{\mathbb{N}} \left\{ \begin{array}{l} [] \in V_{\mathbb{N}} \\ \textbf{se } n \in \mathbb{N} \text{ e } v \in V_{\mathbb{N}}, \text{ então } \textbf{append}(n, v) \in V_{\mathbb{N}} \end{array} \right.$$

$$f: V_{\mathbb{N}} \rightarrow V_{\mathbb{N}} \quad \textbf{head} : \mathbb{N}^n \rightarrow \mathbb{N} \quad \textbf{head}(v) = v_1$$
$$f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)]$$

$$v = [v_1, v_2, \dots, v_n] \in \mathbb{N}^n$$

$$\mathbf{append} : \mathbb{N} \times \mathbb{N}^n \rightarrow \mathbb{N}^{n+1} \quad \mathbf{append}(n, v) = [x, v_1, \dots, v_n]$$

$$V_{\mathbb{N}} \left\{ \begin{array}{l} [] \in V_{\mathbb{N}} \\ \text{se } n \in \mathbb{N} \text{ e } v \in V_{\mathbb{N}}, \text{ então } \mathbf{append}(n, v) \in V_{\mathbb{N}} \end{array} \right.$$

$$\begin{aligned} f &: V_{\mathbb{N}} \rightarrow V_{\mathbb{N}} \\ f([v_1, v_2, \dots, v_n]) &= [h(v_1), h(v_2), \dots, h(v_n)] \end{aligned}$$

$$\begin{aligned} \mathbf{head} &: \mathbb{N}^n \rightarrow \mathbb{N} \quad \mathbf{head}(v) = v_1 \\ \mathbf{tail} &: \mathbb{N}^n \rightarrow \mathbb{N}^{n-1} \quad \mathbf{tail}(v) = [v_2, \dots, v_n] \end{aligned}$$

$$v = [v_1, v_2, \dots, v_n] \in \mathbb{N}^n \quad \textbf{append} : \mathbb{N} \times \mathbb{N}^n \rightarrow \mathbb{N}^{n+1} \quad \textbf{append}(n, v) = [x, v_1, \dots, v_n]$$

$$V_{\mathbb{N}} \left\{ \begin{array}{l} [] \in V_{\mathbb{N}} \\ \textbf{se } n \in \mathbb{N} \text{ e } v \in V_{\mathbb{N}}, \text{ então } \textbf{append}(n, v) \in V_{\mathbb{N}} \end{array} \right.$$

$$\begin{array}{ll} f: V_{\mathbb{N}} \rightarrow V_{\mathbb{N}} & \textbf{head} : \mathbb{N}^n \rightarrow \mathbb{N} \quad \textbf{head}(v) = v_1 \\ f([v_1, v_2, \dots, v_n]) = [h(v_1), h(v_2), \dots, h(v_n)] & \textbf{tail} : \mathbb{N}^n \rightarrow \mathbb{N}^{n-1} \quad \textbf{tail}(v) = [v_2, \dots, v_n] \end{array}$$

$$f \left\{ \begin{array}{l} f([]) = [] \\ f(v) = \textbf{append} \left(h \left(\textbf{head}(v) \right), f \left(\textbf{tail}(v) \right) \right) \end{array} \right.$$

Voltando à programação, podemos redefinir nossa função com uma sintaxe mais parecida à matemática

```
def f(v) do
  if size(v) = 0 do
    return []
  else
    first = head(v)
    rest = tail(v)
    return h(first) ++ f(rest)
  end
end
```

Voltando à programação, podemos redefinir nossa função com uma sintaxe mais parecida à matemática

```
def f(v) do
  if size(v) = 0 do
    return []
  else
    first = head(v)
    rest = tail(v)
    return h(first) ++ f(rest)
  end
end
```

```
def f([]) do [] end
def f(v) do
  first = head(v)
  rest = tail(v)
  return h(first) ++ f(rest)
end
```

Voltando à programação, podemos redefinir nossa função com uma sintaxe mais parecida à matemática

```
def f(v) do
  if size(v) = 0 do
    return []
  else
    first = head(v)
    rest = tail(v)
    return h(first) ++ f(rest)
  end
end
```

Definição sem pattern matching

```
def f([]) do [] end
def f(v) do
  first = head(v)
  rest = tail(v)
  return h(first) ++ f(rest)
end
```

Definição com pattern matching

Pattern matching é uma funcionalidade muito poderosa e comum nas linguagens funcionais

```
def f(1) do ... end
```

```
def f(2) do ... end
```

```
def f(n) when is_integer(n) do ... end
```

```
def f([]) do ... end
```

```
def f([a]) do ... end
```

```
def f([head | tail]) do ... end
```

```
def f(x) do ... end
```

Pattern matching é uma funcionalidade muito poderosa e comum nas linguagens funcionais

```
def f(1) do ... end  
def f(2) do ... end  
def f(n) when is_integer(n) do ... end  
def f([]) do ... end  
def f([a]) do ... end  
def f([head | tail]) do ... end  
def f(x) do ... end
```

O interpretador avalia as definições de cima para baixo e para na primeira coincidência

Pattern matching é uma funcionalidade muito poderosa e comum nas linguagens funcionais

```
def f(1) do ... end  
def f(2) do ... end  
def f(n) when is_integer(n) do ... end  
def f([]) do ... end  
def f([a]) do ... end  
def f([head | tail]) do ... end  
def f(x) do ... end
```

O interpretador avalia as definições de cima para baixo e para na primeira coincidência

Tiramos as estruturas condicionais do código e colocamos na definição das funções

**Na programação funcional também nos apoiamos
sobre outras duas propriedades**

**Na programação funcional também nos apoiamos
sobre outras duas propriedades**

Imutabilidade dos dados

**Na programação funcional também nos apoiamos
sobre outras duas propriedades**

Imutabilidade dos dados

**Sem
variáveis intermediárias**

Na programação funcional também nos apoiamos sobre outras duas propriedades

Imutabilidade dos dados

Sem
variáveis intermediárias

Pureza das funções

**Na programação funcional também nos apoiamos
sobre outras duas propriedades**

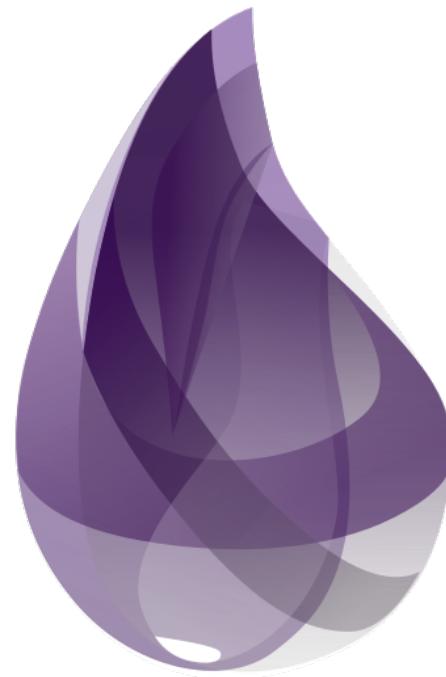
Imutabilidade dos dados

**Sem
variáveis intermediárias**

Pureza das funções

Vamos ir entendendo esses conceitos à medida que avancemos

Vamos começar com Elixir

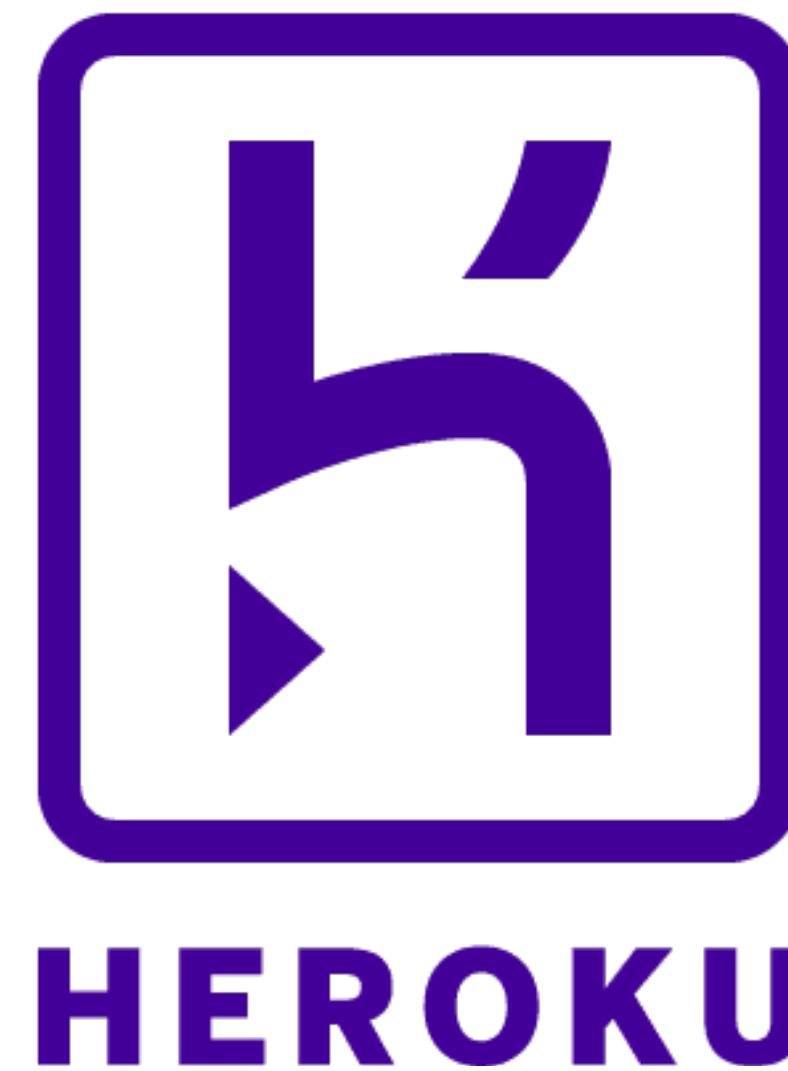


Quem usa Elixir?

Quem usa Elixir?



Quem usa Elixir?



Quem usa Elixir?



Elixir é uma linguagem compilada

Elixir é uma linguagem compilada

(e)ERLANG

Elixir é uma linguagem compilada



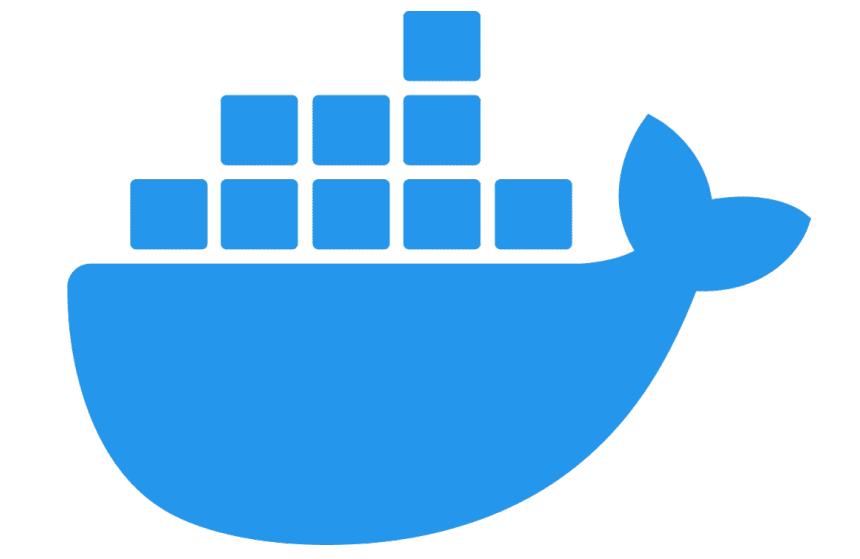
Se executa sobre uma máquina virtual, a BEAM

Elixir é uma linguagem compilada

(e)ERLANG

Se executa sobre uma máquina virtual, a BEAM
É rápida e extremamente boa para concorrência, o que a faz escalável

<https://github.com/andres-vidal/elixir-course>



docker[®]

Máquina virtual com o
ambiente de desenvolvimento



GNU Make

Executar os comandos
relevantes facilmente

<https://github.com/andres-vidal/elixir-course>

```
make install # baixa o ambiente  
make run      # conecta o terminal  
mix test       # executa os testes
```

No Windows sem make

executar uma vez

```
docker run --name elixir -v %cd%:/workspace -d -t elixir:latest
```

executar para abrir o terminal

```
docker exec -it elixir bash
```

Elixir Web Console

<https://elixirconsole.wyeworks.com/>

Elixir é uma linguagem funcional, mas não é pura!

Elixir é uma linguagem funcional, mas não é pura!

Estruturas condicionais

Elixir é uma linguagem funcional, mas não é pura!

Estruturas condicionais

Temos variáveis
intermediárias

Elixir é uma linguagem funcional, mas não é pura!

Estruturas condicionais

Temos variáveis intermediárias

Funções podem não ser puras

Elixir tem tipado dinâmico e suporte para varios tipos

Number

`0, 1, 2, 3`

`0.1, 0.2, 0.3`

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3

0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3

0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]

[{:a, 1}, {:b, 2}, {:c, 3}]

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]
[{:a, 1}, {b: 2}, {c: 3}]

Map

%{a: 1, b: 2, c: 3}

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]
[{:a, 1}, {b: 2}, {c: 3}]

Map

%{a: 1, b: 2, c: 3}
%{:a => 1, :b => 2, :c => 3}

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]
[{:a, 1}, {b: 2}, {c: 3}]

Map

%{a: 1, b: 2, c: 3}
%{:a => 1, :b => 2, :c => 3}
%{"a" => 1, "b" => 2, "c" => 3}

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]
[{:a, 1}, {b: 2}, {c: 3}]

Map

%{a: 1, b: 2, c: 3}
%{:a => 1, :b => 2, :c => 3}
%{"a" => 1, "b" => 2, "c" => 3}

Structs

Elixir tem tipado dinâmico e suporte para varios tipos

Number

0, 1, 2, 3
0.1, 0.2, 0.3

Binary

“Olá, Mundo!”

Atom

:ola_mundo

Tuple

{0, 1, 2, 3}

List

[0, 1, 2, 3]

Keyword

[a: 1, b: 2, c: 3]
[{:a, 1}, {b: 2}, {c: 3}]

Map

%{a: 1, b: 2, c: 3}
%{:a => 1, :b => 2, :c => 3}
%{"a" => 1, "b" => 2, "c" => 3}

Structs

nil

Elixir tem tipado dinâmico e suporte para varios tipos

Number

`0, 1, 2, 3
0.1, 0.2, 0.3`

Binary

`“Olá, Mundo!”`

Atom

`:ola_mundo`

Tuple

`{0, 1, 2, 3}`

List

`[0, 1, 2, 3]`

Keyword

`[a: 1, b: 2, c: 3]
[{:a, 1}, {b: 2}, {c: 3}]`

Map

`%{a: 1, b: 2, c: 3}
%{:a => 1, :b => 2, :c => 3}
%{“a” => 1, “b” => 2, “c” => 3}`

Structs

nil

Elixir tem operadores básicos para manipular dados

Elixir tem operadores básicos para manipular dados

Aritméticos

`+, -, *, /,`
`div, rem`

Elixir tem operadores básicos para manipular dados

Aritméticos
+, -, *, /,
div, rem

Relacionais
>, >=, <, <=
==, ===,
!=, !==

Elixir tem operadores básicos para manipular dados

Aritméticos
+, -, *, /,
div, rem

Relacionais
>, >=, <, <=
==, ===,
!=, !==

Lógicos
and, or not
&&, ||, !

Elixir tem operadores básicos para manipular dados

Aritméticos
+, -, *, /,
div, rem

Relacionais
>, >=, <, <=
==, ===,
!=, !==

Lógicos
and, or not
&&, ||, !

Concatenação
de Strings
<>

Elixir tem operadores básicos para manipular dados

Aritméticos
+, -, *, /,
div, rem

Relacionais
>, >=, <, <=
==, ===,
!=, !==

Lógicos
and, or not
&&, ||, !

Concatenação
de Strings
<>

Concatenação
de Listas
++

Elixir tem operadores básicos para manipular dados

Aritméticos
+, -, *, /,
div, rem

Relacionais
>, >=, <, <=
==, ===,
!=, !==

Lógicos
and, or not
&&, ||, !

Concatenação
de Strings
<>

Concatenação
de Listas
++

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do [] end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do [] end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do [] end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Com a palavra **defp** podemos definir funções privadas

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do [] end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Com a palavra **defp** podemos definir funções privadas

Blocos de código são delimitados pelas palavras reservadas **do** e **end**

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do []
  end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Com a palavra **defp** podemos definir funções privadas

Blocos de código são delimitados pelas palavras reservadas **do** e **end**

Não precisamos **return** para retornar de uma função, sempre se utilizará a última linha do bloco

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do []
  end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Com a palavra **defp** podemos definir funções privadas

Blocos de código são delimitados pelas palavras reservadas **do** e **end**

Não precisamos **return** para retornar de uma função, sempre se utilizará a última linha do bloco

Não precisamos parênteses para chamar funções

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do []
  end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end

  defp h(x) do
    x+1
  end
end

MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Com a palavra **defp** podemos definir funções privadas

Blocos de código são delimitados pelas palavras reservadas **do** e **end**

Não precisamos **return** para retornar de uma função, sempre se utilizará a última linha do bloco

Não precisamos parênteses para chamar funções

Chamamos funções do mesmo módulo pelo nome e funções de outros módulos utilizando a notação **Modulo.nome**

Elixir organiza funções em módulos

```
defmodule MeuModulo do
  def f([]) do []
  end
  def f([head | tail]) do
    h(head) ++ f(tail)
  end
```

```
  defp h(x) do
    x+1
  end
end
```

```
MeuModulo.f [1, 2, 3]
# => [2, 3, 4]
```

Com a palavra **def** podemos definir funções públicas

Com a palavra **defp** podemos definir funções privadas

Blocos de código são delimitados pelas palavras reservadas **do** e **end**

Não precisamos **return** para retornar de uma função, sempre se utilizará a última linha do bloco

Não precisamos parênteses para chamar funções

Chamamos funções do mesmo módulo pelo nome e funções de outros módulos utilizando a notação **Modulo.nome**

A linguagem vem com uma série de módulos com funções úteis para trabalhar com as estruturas de dados disponíveis

Elixir tem funções como cidadãos de primeira classe

```
f = &MeuModulo.f/1
```

Elixir tem funções como cidadãos de primeira classe

```
f = &MeuModulo.f/1
```

```
f.([1, 2, 3])  
# => [2, 3, 4]
```

Elixir tem funções como cidadãos de primeira classe

```
f = &MeuModulo.f/1
```

```
f.([1, 2, 3])  
# => [2, 3, 4]
```

```
soma_um = fn n -> n + 1 end
```

```
soma_um.(1)  
# => 2
```

Elixir tem pattern matching

a = [1, 2, 3]

[a, b, c] = [1, 2, 3]

[head | tail] = [1, 2, 3]

“Olá, ” <> mundo = “Olá, Mundo”

{a, b, c} = {1, 2, 3}

%{a: a, b: b, c: c} = %{a: 1, b: 2, c: 3}

Elixir tem estruturas condicionais

```
type = case x do
    []  -> "lista vazia"
    [a] -> "lista com um elemento"
    1  -> "numero um"
    n when n < 0 -> "numero negativo"
    n when is_number(n) -> "outro numero"
    "olá, mundo" -> "string olá, mundo"
    nil -> "nulo"
    %{ } -> "mapa"
    %{a: 1} -> "mapa com valor 1 na chave a"
end
```

Elixir tem estruturas condicionais

```
type = if is_number(n) do
    "number"
else
    "other"
end
```

Elixir tem estruturas condicionais

```
type = if is_number(n) do
    "number"
else
    "other"
end
```

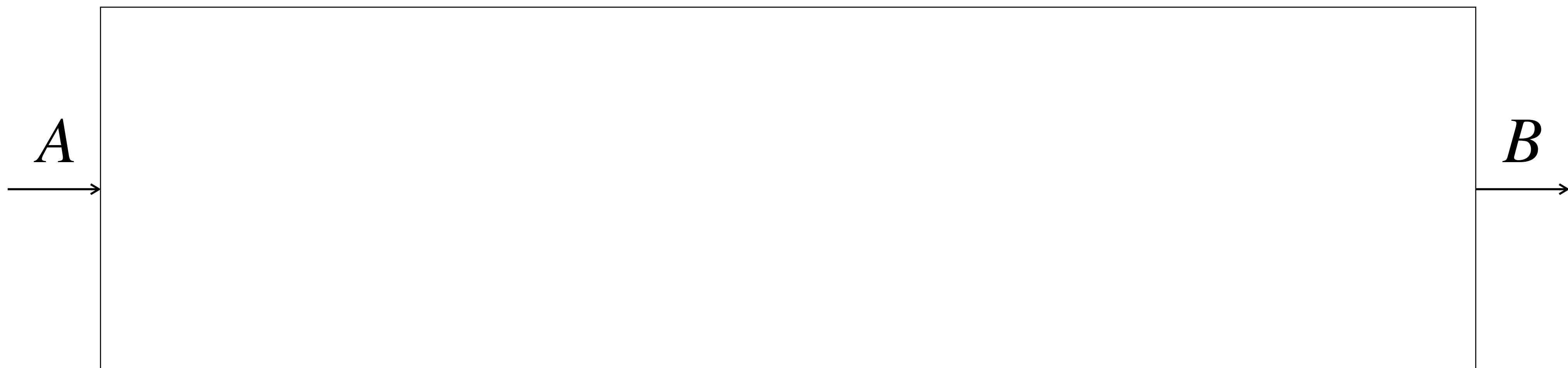
```
type = unless is_number(n) do
    "other"
else
    "number"
end
```

Elixir tem estruturas condicionais

```
type = cond do
    x == 1 -> "numero um"
    is_number(x) -> "numero"
    is_list(x) and length(x) == 0 -> "lista vazia"
    true -> "outro"
end
```

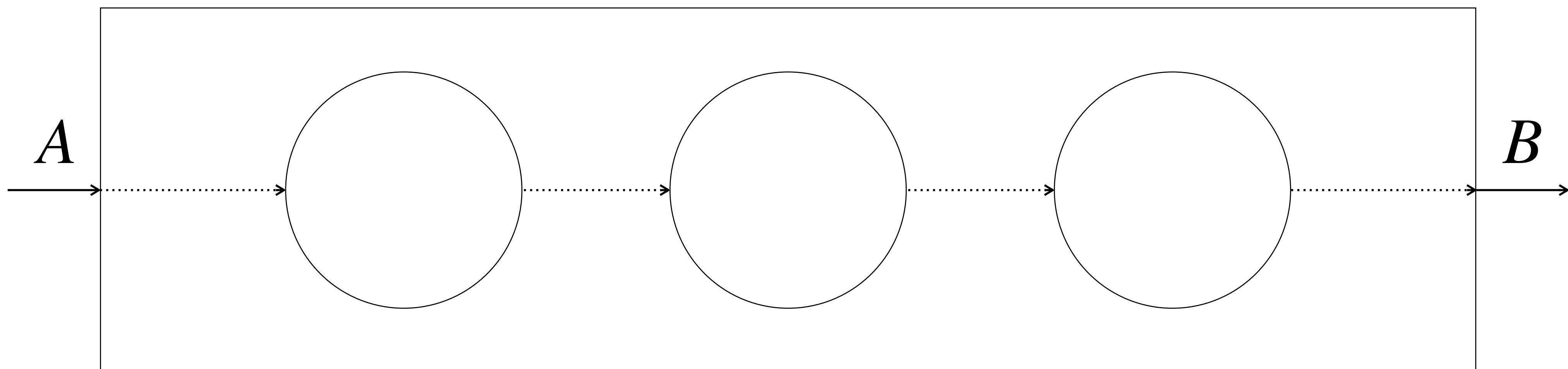
Elixir tem pipelining

$$p : A \rightarrow B$$

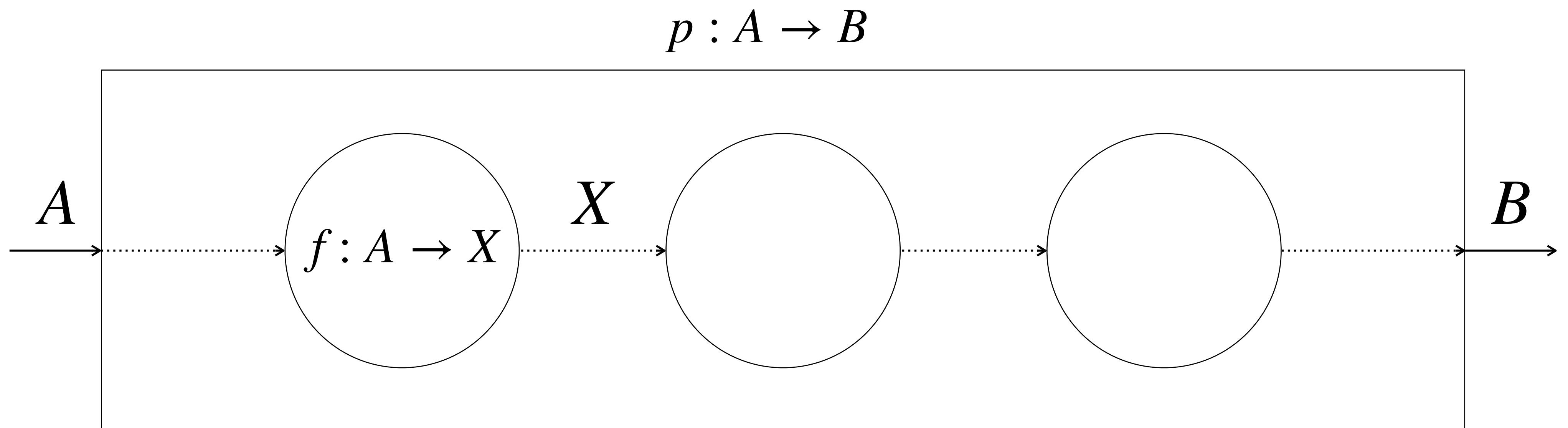


Elixir tem pipelining

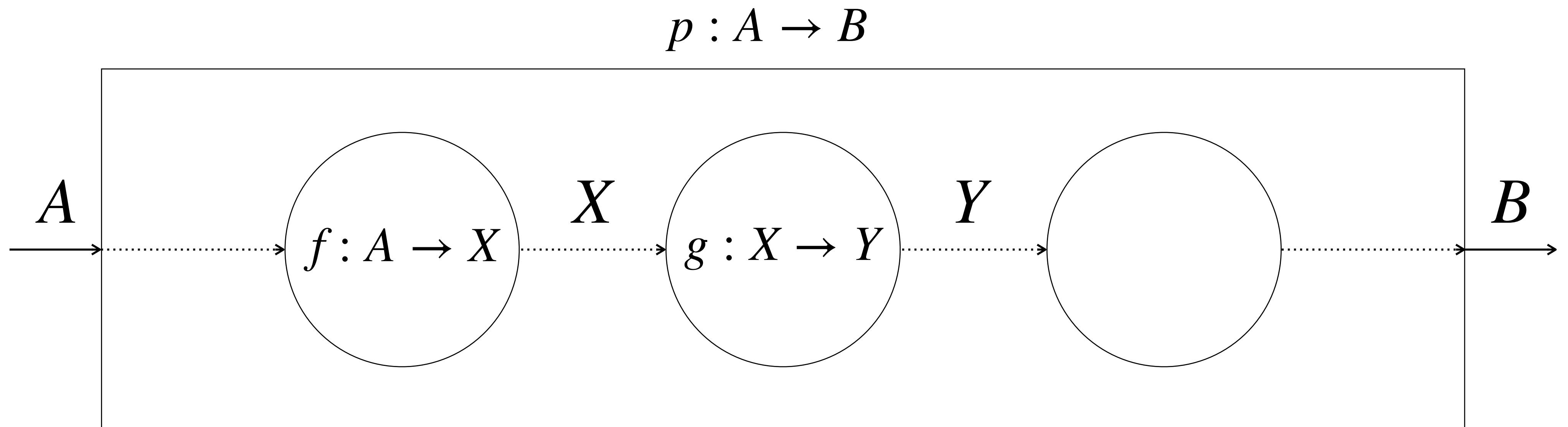
$$p : A \rightarrow B$$



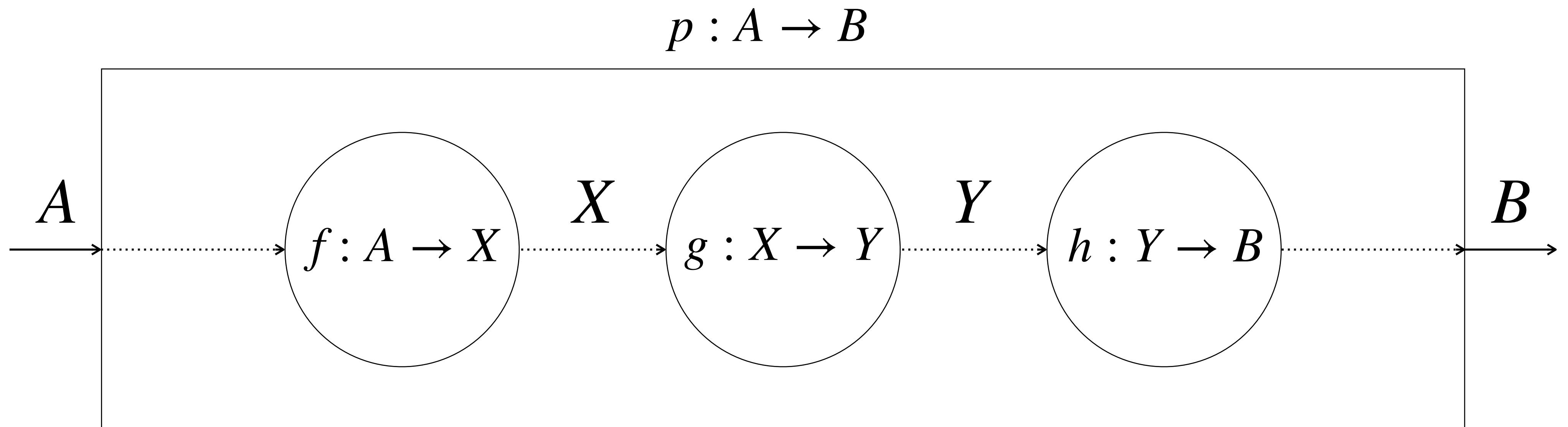
Elixir tem pipelining



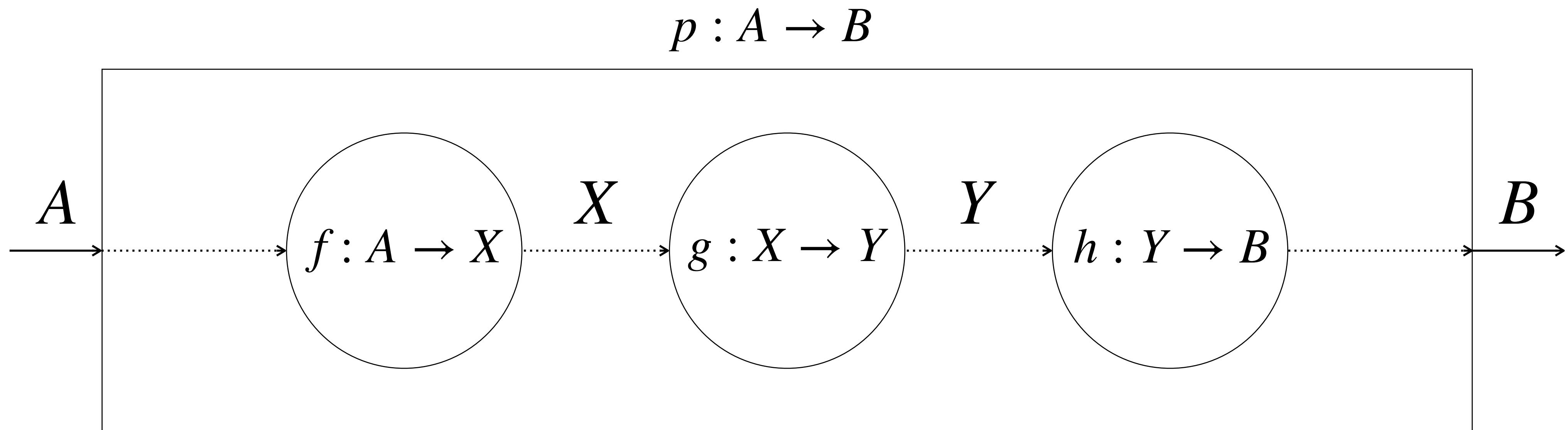
Elixir tem pipelining



Elixir tem pipelining

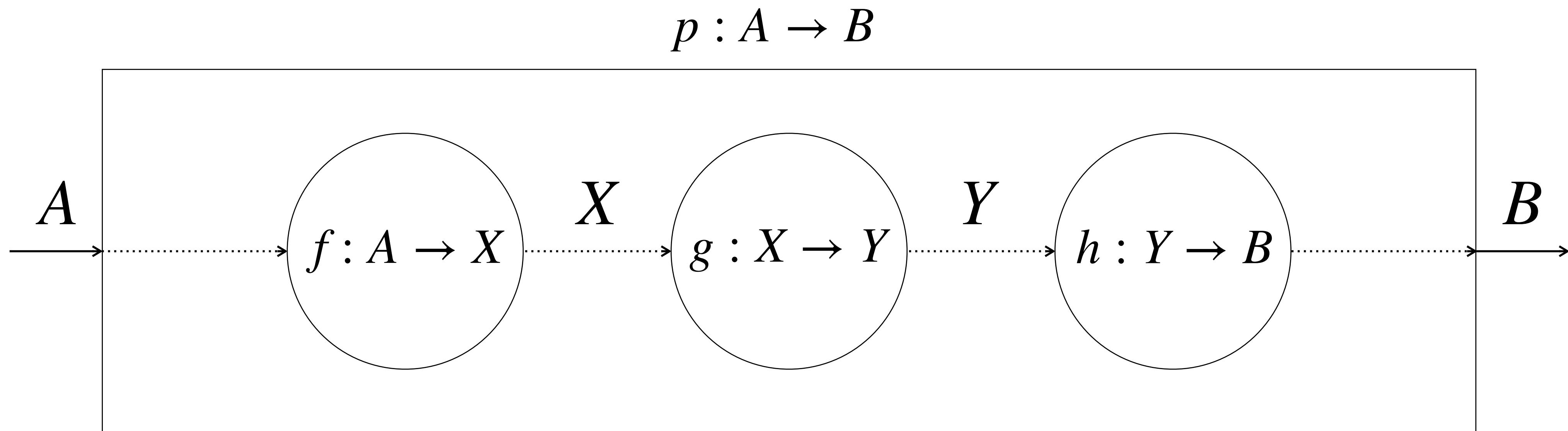


Elixir tem pipelining



```
def p(a) do
  h(g(f(a)))
end
```

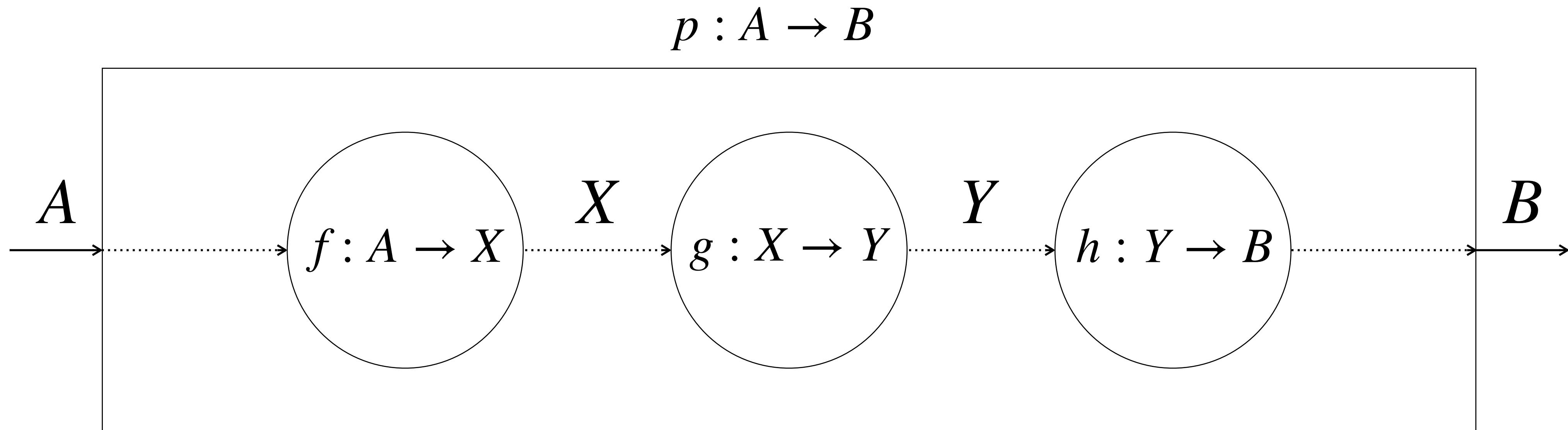
Elixir tem pipelining



```
def p(a) do
  h(g(f(a)))
end
```

```
def p(a) do
  x = f(a)
  y = g(x)
  h(y)
end
```

Elixir tem pipelining



```
def p(a) do
  h(g(f(a)))
end
```

```
def p(a) do
  x = f(a)
  y = g(x)
  h(y)
end
```

```
def p(a) do
  a
  |> f
  |> g
  |> h
end
```

Na matemática temos composição de funções

$$p(a) = (h \circ g \circ f)(a)$$

Na matemática temos composição de funções

$$p(a) = (h \circ g \circ f)(a)$$

def p(a) do a |> f |> g |> h end

Na matemática temos composição de funções

$$p(a) = (h \circ g \circ f)(a)$$

def p(a) do a |> f |> g |> h end

Que é o pipe operator lido de direita para esquerda

Operações de Manipulação de Estruturas de Dados

Map, Filter, Reduce

Map

Recebe um vetor v e uma função de mapping m .

Aplica m em cada elemento do vetor v e retorna o vetor resultante.

$$\mathbf{map}(v, m) = \mathbf{map}([v_1, v_2, \dots, v_n], m) = [m(v_1), m(v_2), \dots, m(v_n)]$$

$$\mathbf{map}(v, m) = [m(v_i) : v_i \in v]$$

Map

Se queremos somar 1 a todos os elementos de um vetor:

$$m(x) = x + 1$$

$$\mathbf{map}(v, m) = \mathbf{map}([v_1, v_2, \dots, v_n], m) = [v_1 + 1, v_2 + 1, \dots, v_n + 1]$$

```
Enum.map([1, 2, 3, 4, 5], fn x -> x + 1 end)  
# => [2, 3, 4, 5, 6]
```

Filter

Recebe um vetor v e uma função de filtro f que retorna true ou false
Retorna um vetor com os elementos para os quais f retorna true

$$\mathbf{filter}(v, m) = [v_i : v_i \in v \wedge f(v_i) = \text{true}]$$

Filter

Se queremos ficar só com os números positivos de um vetor

$$f(x) = x \geq 0$$

$$\mathbf{filter}(\nu, m) = [\nu_i : \nu_i \in \nu \wedge \nu_1 \geq 0]$$

```
Enum.filter([1, -2, 3, -4, 5], fn x -> x >= 0 end)  
# => [1, 3, 5]
```

Reduce

Map e **Filter** são operações que processam elemento a elemento sem informação do processamento dos demais

Reduce introduz a possibilidade de transmitir informação para o seguinte passo a través de um **valor acumulado**

Reduce

Contar os elementos de um vetor

[a b c d e]

Reduce

Contar os elementos de um vetor

[a b c d e]

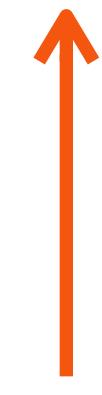
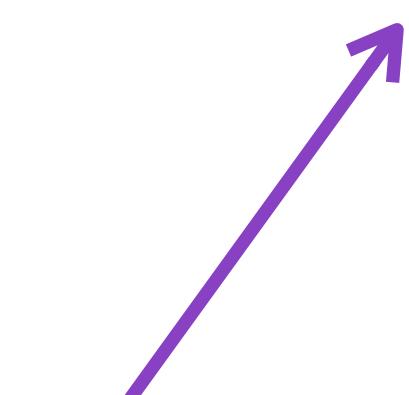
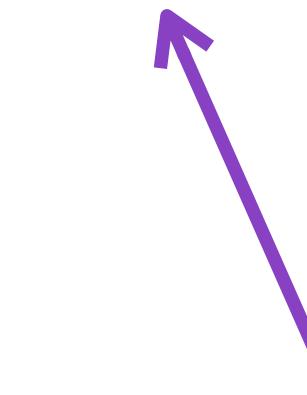
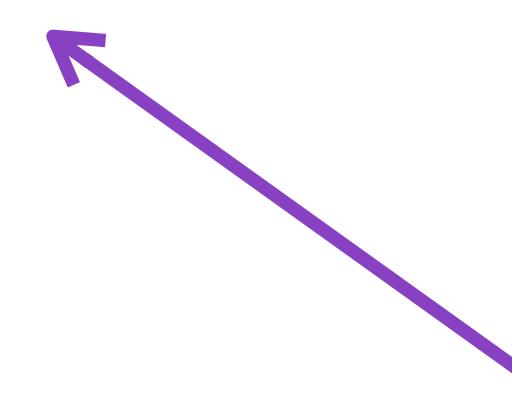
[{1, a} {2, b} {3, c} {4, d} {5, e}]

Reduce

Contar os elementos de um vetor

[a b c d e]

[{1, a} {2, b} {3, c} {4, d} {5, e}]



Resultados Intermediarios

Resultado Final

Reduce

Contar os elementos de um vetor

[{**0**, a} ...]

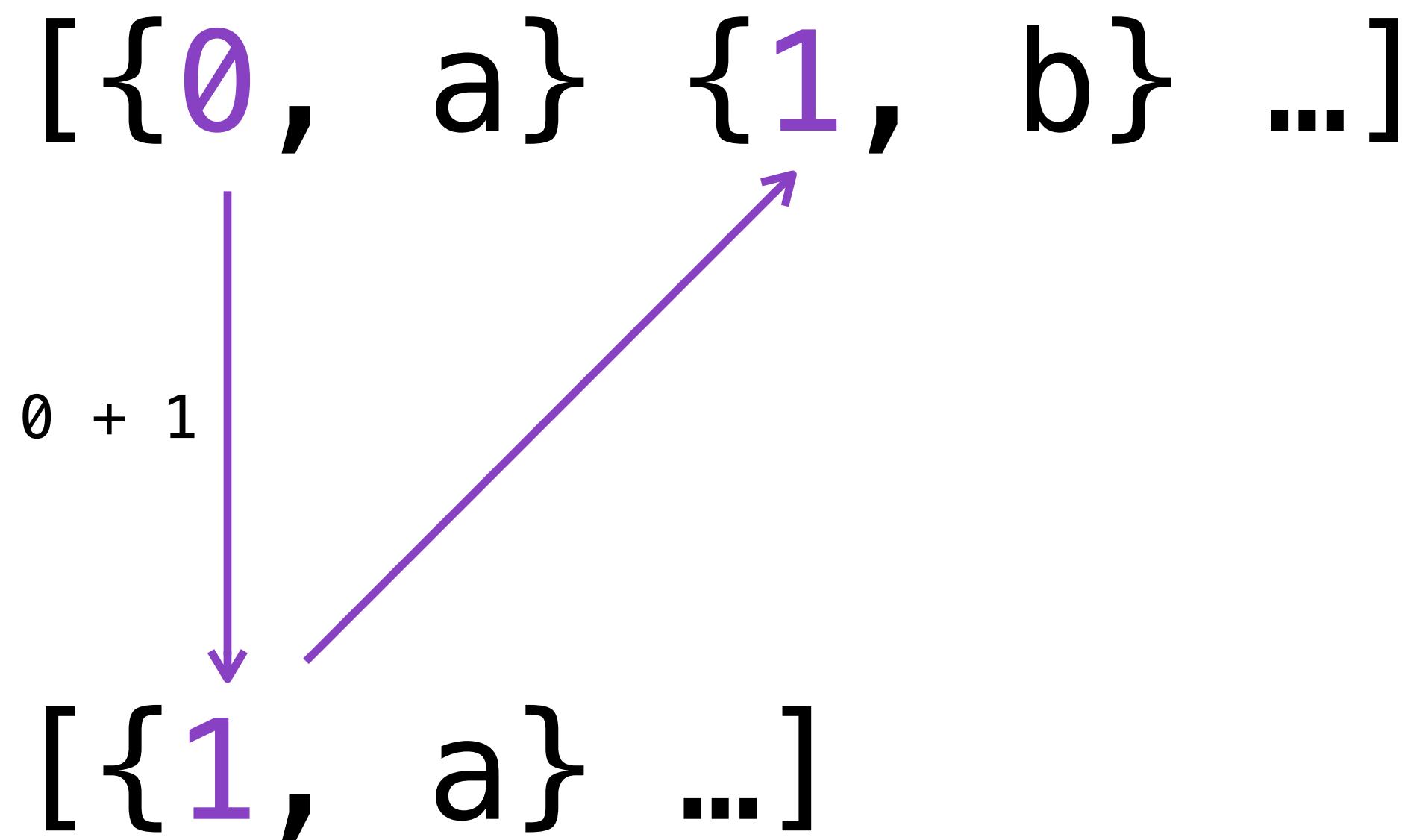
0 + 1



[{**1**, a} ...]

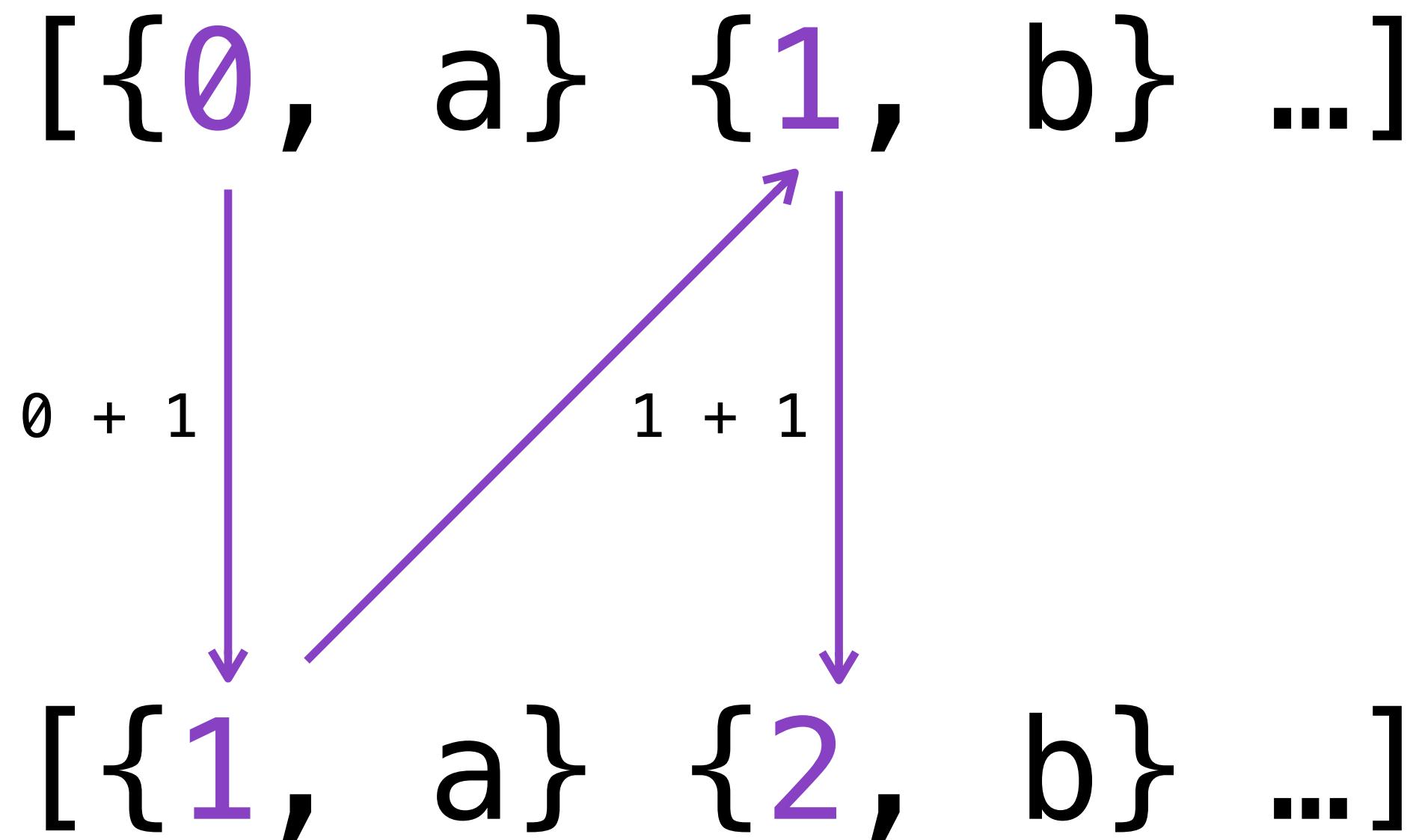
Reduce

Contar os elementos de um vetor



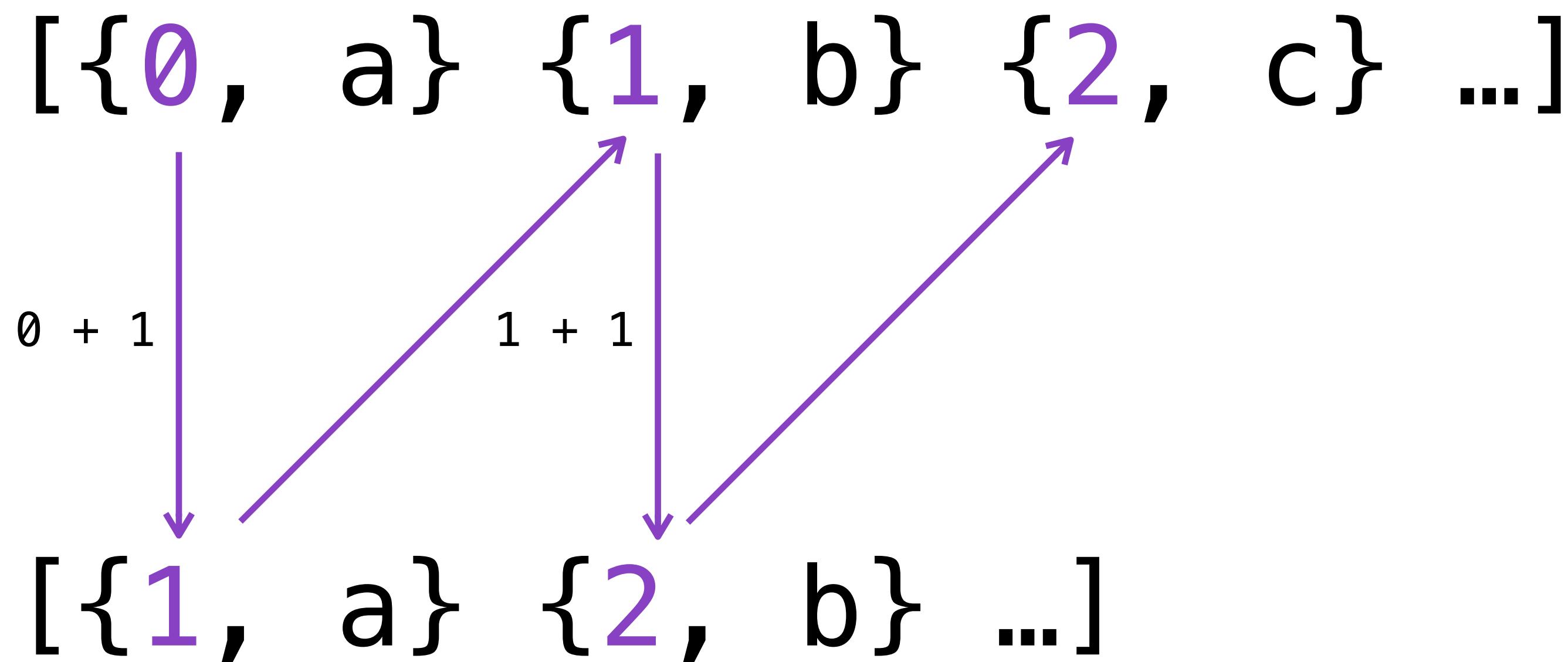
Reduce

Contar os elementos de um vetor



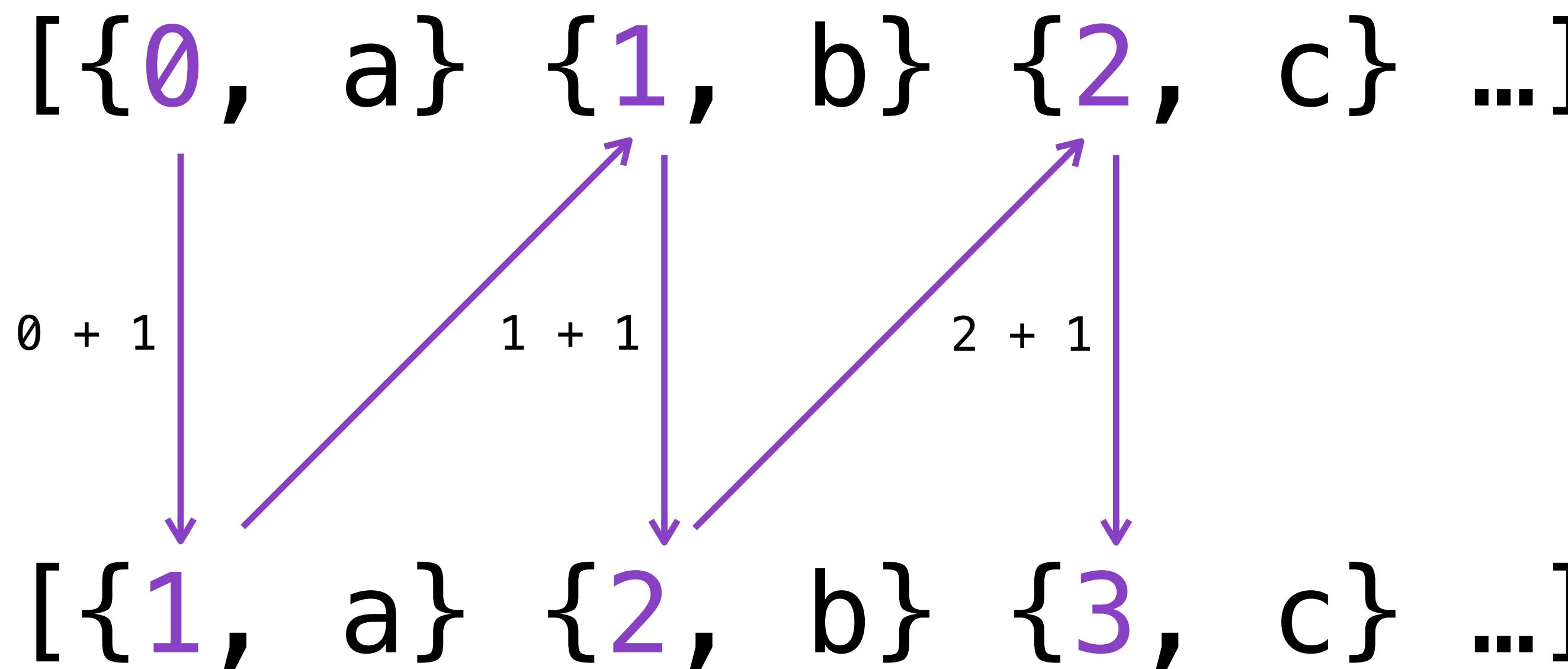
Reduce

Contar os elementos de um vetor



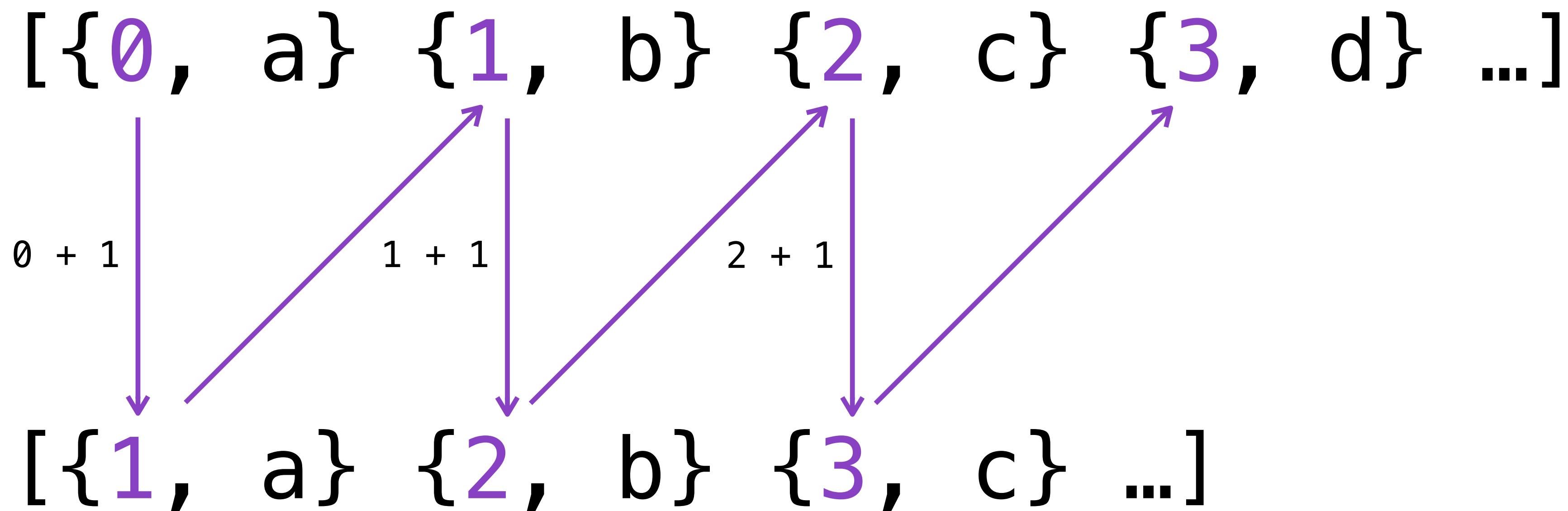
Reduce

Contar os elementos de um vetor



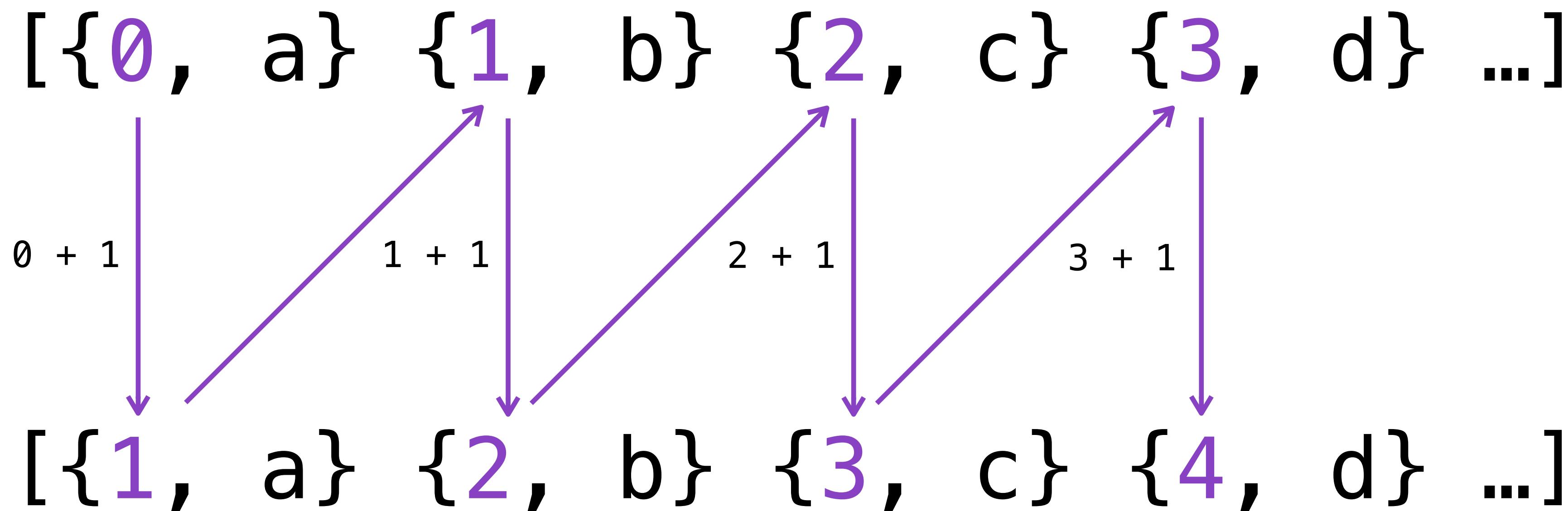
Reduce

Contar os elementos de um vetor



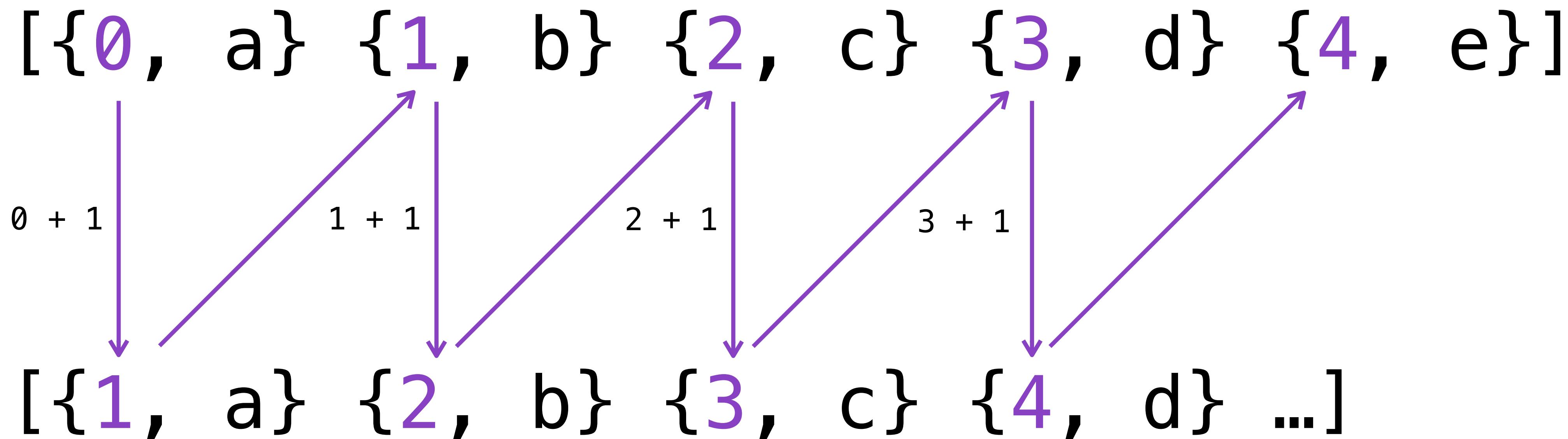
Reduce

Contar os elementos de um vetor



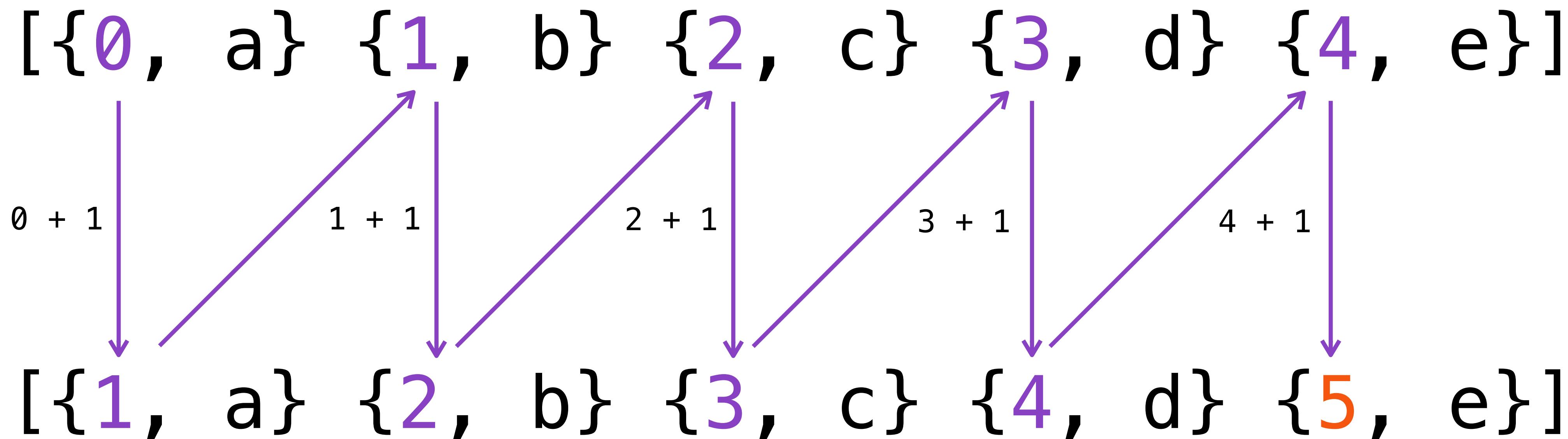
Reduce

Contar os elementos de um vetor



Reduce

Contar os elementos de um vetor



Reduce

Pensando em genérico, precisamos definir uma função que calcule o próximo valor acumulado em base ao anterior e ao elemento atual.

Também precisamos de um valor inicial para utilizar no primeiro passo.

Reduce

```
# Contar os elementos de um vetor  
Enum.reduce([1, 2, 3, 4, 5], 0, fn acc, e -> acc + 1 end)  
# => 5
```

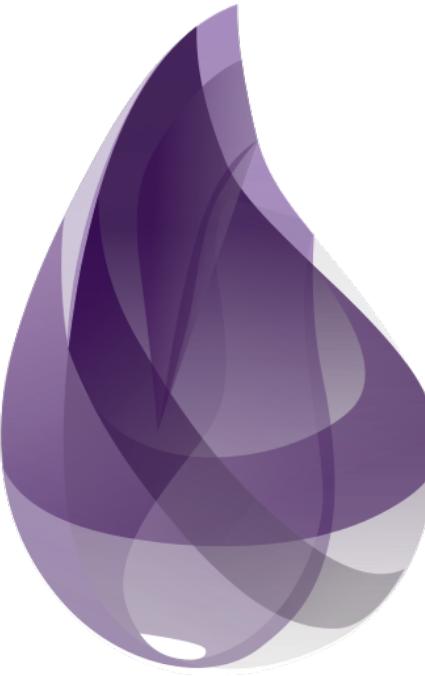
Reduce

```
# Somar os elementos de um vetor  
Enum.reduce([1, 1, 1, 1, 1], 0, fn acc, e -> acc + e end)  
# => 5
```

Reduce

Recebe
um vetor **v**,
um valor inicial **i**
função **f**
que recebe um **valor acumulado** e um **elemento** de **v**
e retorna o valor acumulado após processar o elemento
Retorna um vetor com os elementos para os quais **f** retorna true

Criando uma lista de tarefas



Vamos criar uma lista de tarefas com os seguintes requisitos:

./tarefas

lista as tarefas que ainda não foram completadas

./tarefas todas

lista todas as tarefas

as completadas aparecem no final com o subtítulo – **Tarefas Completadas** –

./tarefas <descrição>

cadastra uma tarefa nova na lista de tarefas, com a **descrição** fornecida
a tarefa é cadastrada na última posição

./tarefas <descrição> <posição>

cadastra uma tarefa nova na lista de tarefas, com a **descrição** e na **posição** fornecidas

Vamos criar uma lista de tarefas com os seguintes requisitos:

./tarefas mover <posição_origem> <posição_destino>
muda a posição da tarefa na **posição_origem** para a **posição_destino**

./tarefas remover <posição>
remove a tarefa da **posição** fornecida da lista de tarefas

./tarefas completar <posição>
marca a tarefa da **posição** fornecida da lista como completada

O que vamos precisar?

Uma representação para os dados
{descrição, estado}

Um lugar onde salvar os dados
tarefas.txt

```
mix new <nome>      # cria um projeto novo  
mix escript.build # cria um executável  
mix test           # executa os testes
```

mix escript.build e **mix test** devem ser executados dentro do diretório do projeto
mix escript.build requer adicionar `main_module: <modulo com o método main>` na
seção “project” do `mix.exs`

Modelado de dados com Ecto



Interface web com Phoenix





<https://chat.whatsapp.com/C2umv7ApJ7F4FoZF40sgib>