Funções como valores

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

As principais características que vimos até agora do paradigma funcional foram:

- Ausência de efeitos colaterais;
- Tipos algébricos e autorreferências;
- Casamento de padrões;
- Recursão como forma de especificar iteração.

Veremos a seguir outra característica essencial do paradigma funcional: funções como valores.

Funções são **entidades de primeira classe** se:

- Podem ser usadas, sem restrições, onde outros valores podem ser usados (passado como entrada, devolvido como resultado, armazenado em listas, etc);
- Podem ser construídas, sem restrições, onde outros valores também podem (localmente, em expressões, etc);
- Podem ser tipadas de forma similar a outro valores (existe um tipo associado com cada função e esse tipo pode ser usado para compor outro tipos).

Veremos a seguir como as funções podem ser utilizadas como valores.

Uma **função de alta ordem** é aquela que:

- Recebe como entrada uma ou mais funções; e/ou
- Produz como saída uma ou mais funções.

Funções que recebem funções como parâmetro

Como identificar a necessidade de utilizar funções como parâmetro?

Encontrando similaridades entre funções.

Vamos ver diversas pares de funções e identificar similaridades entre elas.

Por questões de espaço, no restante desse material, usamos `p` para primeiro e `r` para `resto` e colocamos os exemplos fora de funções `_examples`.

Exemplo: `contem_3` e `contem_5`

Vamos começar com um exemplo simples.

Vamos criar uma função que abstrai o comportamento das funções `contem_3` e `contem_5`.

Exemplo: contem_3 e contem_5

```gleam
/// Devolve True se 3 está em *lst*,
/// False caso contrário.
fn contem_3(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 3 || contem_3(r)
  }
}
check.eq(contem_3([4, 3, 1]), True)

/// Devolve True se 5 está em *lst*,
/// False caso contrário.
fn contem_5(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 5 || contem_5(r)
  }
}
check.eq(contem_5([4, 3, 1]), False)
```

Vamos definir uma função que abstrai o comportamento de `contem_3` e `contem_5`.

```gleam
fn contem(lst, n) {
  case lst {
    [] -> False
    [p, ..r] -> p == n || contem(r, n)
  }
}
```

Exemplo: contem_3 e contem_5

```gleam
/// Devolve True se 3 está em *lst*,
/// False caso contrário.
fn contem_3(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 3 || contem_3(r)
  }
}
check.eq(contem_3([4, 3, 1]), True)

/// Devolve True se 5 está em *lst*,
/// False caso contrário.
fn contem_5(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 5 || contem_5(r)
  }
}
check.eq(contem_5([4, 3, 1]), False)
```

Vamos definir uma função que abstrai o comportamento de `contem_3` e `contem_5`.

```gleam
fn contem(lst, n) {
  case lst {
    [] -> False
    [p, ..r] -> p == n || contem(r, n)
  }
}

check.eq(contem([4, 3, 1], 3), True)
check.eq(contem([4, 3, 1], 5), False)
```

Exemplo: contem_3 e contem_5

```
/// Devolve True se 3 está em *lst*,
/// False caso contrário.
fn contem_3(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 3 || contem_3(r)
  }
}
check.eq(contem_3([4, 3, 1]), True)

/// Devolve True se 5 está em *lst*,
/// False caso contrário.
fn contem_5(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 5 || contem_5(r)
  }
}
check.eq(contem_5([4, 3, 1]), False)
```

Vamos definir uma função que abstrai o comportamento de `contem_3` e `contem_5`.

```
/// Devolve True se *n* está em *lst*,
/// False caso contrário.
fn contem(lst, n) {
  case lst {
    [] -> False
    [p, ..r] -> p == n || contem(r, n)
  }
}

check.eq(contem([4, 3, 1], 3), True)
check.eq(contem([4, 3, 1], 5), False)
```

Exemplo: `contem_3` e `contem_5`

```gleam
/// Devolve True se 3 está em *lst*,
/// False caso contrário.
fn contem_3(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 3 || contem_3(r)
  }
}
check.eq(contem_3([4, 3, 1]), True)

/// Devolve True se 5 está em *lst*,
/// False caso contrário.
fn contem_5(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 5 || contem_5(r)
  }
}
check.eq(contem_5([4, 3, 1]), False)
```

Vamos definir uma função que abstrai o comportamento de `contem_3` e `contem_5`.

```gleam
/// Devolve True se *n* está em *lst*,
/// False caso contrário.
fn contem(lst: List(Int), n: Int) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == n || contem(r, n)
  }
}

check.eq(contem([4, 3, 1], 3), True)
check.eq(contem([4, 3, 1], 5), False)
```

Exemplo: contem_3 e contem_5

```gleam
/// Devolve True se 3 está em *lst*,
/// False caso contrário.
fn contem_3(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 3 || contem_3(r)
  }
}
check.eq(contem_3([4, 3, 1]), True)

/// Devolve True se 5 está em *lst*,
/// False caso contrário.
fn contem_5(lst: List(Int)) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == 5 || contem_5(r)
  }
}
check.eq(contem_5([4, 3, 1]), False)
```

Vamos definir uma função que abstrai o comportamento de contem_3 e contem_5.

```gleam
/// Devolve True se *n* está em *lst*,
/// False caso contrário.
fn contem(lst: List(a), n: a) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == n || contem(r, n)
  }
}

check.eq(contem([4, 3, 1], 3), True)
check.eq(contem([4, 3, 1], 5), False)
```

Exemplo: contem_3 e contem_5

```gleam
/// Devolve True se 3 está em *lst*,
/// False caso contrário.
fn contem_3(lst: List(Int)) -> Bool {
  contem(lst, 3)
}



check.eq(contem_3([4, 3, 1]), True)

/// Devolve True se 5 está em *lst*,
/// False caso contrário.
fn contem_5(lst: List(Int)) -> Bool {
  contem(lst, 5)
}



check.eq(contem_5([4, 3, 1]), False)
```

Vamos definir uma função que abstrai o comportamento de contem_3 e contem_5.

```gleam
/// Devolve True se *n* está em *lst*,
/// False caso contrário.
fn contem(lst: List(a), n: a) -> Bool {
  case lst {
    [] -> False
    [p, ..r] -> p == n || contem(r, n)
  }
}

check.eq(contem([4, 3, 1], 3), True)
check.eq(contem([4, 3, 1], 5), False)
```

Receita para criar abstração a partir de exemplos

1. Identificar funções com corpo semelhante

- Identificar o que muda
- Criar parâmetros para o que muda
- Copiar o corpo e substituir o que muda pelos parâmetros criados

2. Escrever os exemplos

- Reutilizar os exemplos das funções existentes

3. Escrever o propósito
4. Escrever a assinatura
5. Reescrever o código das funções iniciais em termos da nova função

Exemplo: `lista_nega` e `lista_string`

Vamos criar uma função que abstrai o comportamento das funções `lista_nega` e `lista_string`.

Exemplo: lista_nega e lista_string

```gleam
/// Nega cada elemento de *lst*.
fn lista_nega(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] ->
      [int.negate(p), ..lista_nega(r)]
  }
}
check.eq(lista_nega([4, 3]), [-4, -3])

/// Transforma cada elemento de *lst* em string.
fn lista_string(lst: List(Float)) -> List(String) {
  case lst {
    [] -> []
    [p, ..r] ->
      [float.to_string(p), ..lista_string(r)]
  }
}
check.eq(lista_string([3.0, 7.0]), ["3.0", "7.0"])
```

```gleam
fn mapeia(lst, f) {
  case lst {
    [] -> []
    [p, ..r] -> [f(p), ..mapeia(r)]
  }
}
```

Exemplo: lista_nega e lista_string

```gleam
/// Nega cada elemento de *lst*.
fn lista_nega(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] ->
      [int.negate(p), ..lista_nega(r)]
  }
}
check.eq(lista_nega([4, 3]), [-4, -3])

/// Transforma cada elemento de *lst* em string.
fn lista_string(lst: List(Float)) -> List(String) {
  case lst {
    [] -> []
    [p, ..r] ->
      [float.to_string(p), ..lista_string(r)]
  }
}
check.eq(lista_string([3.0, 7.0]), ["3.0", "7.0"])
```

```gleam
fn mapeia(lst, f) {
  case lst {
    [] -> []
    [p, ..r] -> [f(p), ..mapeia(r)]
  }
}

check.eq(
  mapeia([4, 3], int.negate),
  [-4, -3])
check.eq(
  mapeia([3.0, 7.0], float.to_string),
  ["3.0", "7.0"])
```

Exemplo: lista_nega e lista_string

```gleam
/// Nega cada elemento de *lst*.
fn lista_nega(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] ->
      [int.negate(p), ..lista_nega(r)]
  }
}
check.eq(lista_nega([4, 3]), [-4, -3])

/// Transforma cada elemento de *lst* em string.
fn lista_string(lst: List(Float)) -> List(String) {
  case lst {
    [] -> []
    [p, ..r] ->
      [float.to_string(p), ..lista_string(r)]
  }
}
check.eq(lista_string([3.0, 7.0]), ["3.0", "7.0"])
```

```gleam
/// Aplica *f* a cada elemento de *lst*
fn mapeia(lst, f) {
  case lst {
    [] -> []
    [p, ..r] -> [f(p), ..mapeia(r)]
  }
}

check.eq(
  mapeia([4, 3], int.negate),
  [-4, -3])
check.eq(
  mapeia([3.0, 7.0], float.to_string),
  ["3.0", "7.0"])
```

Exemplo: lista_nega e lista_string

```gleam
/// Nega cada elemento de *lst*.
fn lista_nega(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] ->
      [int.negate(p), ..lista_nega(r)]
  }
}
check.eq(lista_nega([4, 3]), [-4, -3])

/// Transforma cada elemento de *lst* em string.
fn lista_string(lst: List(Float)) -> List(String) {
  case lst {
    [] -> []
    [p, ..r] ->
      [float.to_string(p), ..lista_string(r)]
  }
}
check.eq(lista_string([3.0, 7.0]), ["3.0", "7.0"])
```

```gleam
/// Aplica *f* a cada elemento de *lst*
fn mapeia(
  lst: List(a),
  f: fn(a) -> b,
) -> List(b) {
  case lst {
    [] -> []
    [p, ..r] -> [f(p), ..mapeia(r)]
  }
}

check.eq(
  mapeia([4, 3], int.negate),
  [-4, -3])
check.eq(
  mapeia([3.0, 7.0], float.to_string),
  ["3.0", "7.0"])
```

Exemplo: lista_nega e lista_string

```gleam
/// Nega cada elemento de lst*.
fn lista_nega(lst: List(Int)) -> List(Int) {
  mapeia(lst, int.negate)
}
```

```gleam
check.eq(lista_nega([4, 3]), [-4, -3])
```

```gleam
/// Transforma cada elemento de *lst* em string.
fn lista_string(lst: List(Float)) -> List(String) {
  mapeia(lst, float.to_string)
}
```

```gleam
check.eq(lista_string([3.0, 7.0]), ["3.0", "7.0"])
```

```gleam
/// Aplica *f* a cada elemento de *lst*
fn mapeia(
  lst: List(a),
  f: fn(a) -> b,
) -> List(b) {
  case lst {
    [] -> []
    [p, ..r] -> [f(p), ..mapeia(r)]
  }
}
```

```gleam
check.eq(
  mapeia([4, 3], int.negate),
  [-4, -3])
check.eq(
  mapeia([3.0, 7.0], float.to_string),
  ["3.0", "7.0"])
```

map

Como resultado do exemplo anterior obtivemos a função **mapeia**, que é pré-definida em Gleam como `list.map`.

```
> list.map([4, 1, 2], int.is_even)
[True, False, True]
```

```
> list.map([1.2, 3.1], list.wrap)
[[1.2], [3.1]]
```

```
> list.map(["casa", "", "arroz"],
           string.first)
[Ok("c"), Error(Nil), Ok("a)]
```

```
> list.map([[4], [1, 2]], list.length)
[1, 2]
```

Exemplo: `lista_pares` e `lista_nao_vazia`

Vamos criar uma função que abstrai o comportamento das funções `lista_pares` e `lista_nao_vazia`.

```
fn eh_nao_vazia(s: String) -> Bool {
  s != ""
}
```

Exemplo: lista_pares e lista_nao_vazia

```gleam
/// Seleciona os valores pares de *lst*.
fn lista_pares(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] -> case int.is_even(p) {
      True -> [p, ..lista_pares(r)]
      False -> lista_pares(r)
}}}
check.eq(lista_pares([3, 2, 7]), [2])
/// Seleciona as strings não vazias de *lst*.
fn lista_nao_vazia(lst: List(String))
                    -> List(String) {
  case lst {
    [] -> []
    [p, ..r] -> case eh_nao_vazia(p) {
      True -> [p, ..lista_nao_vazia(r)]
      False -> lista_nao_vazia(r)
}}}
check.eq(lista_nao_vazia(["a", "", "b"]),
         ["a", "b"])
```

```gleam
fn filtra(lst, pred) {
  case lst {
    [] -> []
    [p, ..r] -> case pred(p) {
      True -> [p, ..filtra(r, pred)]
      False -> filtra(r, pred)
    }
  }
}
```

Exemplo: lista_pares e lista_nao_vazia

```gleam
/// Seleciona os valores pares de *lst*.
fn lista_pares(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] -> case int.is_even(p) {
      True -> [p, ..lista_pares(r)]
      False -> lista_pares(r)
}}}
check.eq(lista_pares([3, 2, 7]), [2])
/// Seleciona as strings não vazias de *lst*.
fn lista_nao_vazia(lst: List(String))
                    -> List(String) {
  case lst {
    [] -> []
    [p, ..r] -> case eh_nao_vazia(p) {
      True -> [p, ..lista_nao_vazia(r)]
      False -> lista_nao_vazia(r)
}}}
check.eq(lista_nao_vazia(["a", "", "b"]),
         ["a", "b"])
```

```gleam
fn filtra(lst, pred) {
  case lst {
    [] -> []
    [p, ..r] -> case pred(p) {
      True -> [p, ..filtra(r, pred)]
      False -> filtra(r, pred)
    }
  }
}

check.eq(filtra([3, 2, 7], int.is_even), [2])
check.eq(filtra(["a", "", "b"], eh_nao_vazia),
         ["a", "b"])
```

Exemplo: lista_pares e lista_nao_vazia

```gleam
/// Seleciona os valores pares de *lst*.
fn lista_pares(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] -> case int.is_even(p) {
      True -> [p, ..lista_pares(r)]
      False -> lista_pares(r)
}}}
check.eq(lista_pares([3, 2, 7]), [2])
/// Seleciona as strings não vazias de *lst*.
fn lista_nao_vazia(lst: List(String))
                    -> List(String) {
  case lst {
    [] -> []
    [p, ..r] -> case eh_nao_vazia(p) {
      True -> [p, ..lista_nao_vazia(r)]
      False -> lista_nao_vazia(r)
}}}
check.eq(lista_nao_vazia(["a", "", "b"]),
         ["a", "b"])
```

```gleam
/// Seleciona os valores de *lst*
/// para os quais *pred* devolve True.
fn filtra(lst, pred) {
  case lst {
    [] -> []
    [p, ..r] -> case pred(p) {
      True -> [p, ..filtra(r, pred)]
      False -> filtra(r, pred)
    }
  }
}

check.eq(filtra([3, 2, 7], int.is_even), [2])
check.eq(filtra(["a", "", "b"], eh_nao_vazia),
         ["a", "b"])
```

Exemplo: lista_pares e lista_nao_vazia

```gleam
/// Seleciona os valores pares de *lst*.
fn lista_pares(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] -> case int.is_even(p) {
      True -> [p, ..lista_pares(r)]
      False -> lista_pares(r)
}}}
check.eq(lista_pares([3, 2, 7]), [2])
/// Seleciona as strings não vazias de *lst*.
fn lista_nao_vazia(lst: List(String))
                    -> List(String) {
  case lst {
    [] -> []
    [p, ..r] -> case eh_nao_vazia(p) {
      True -> [p, ..lista_nao_vazia(r)]
      False -> lista_nao_vazia(r)
}}}
check.eq(lista_nao_vazia(["a", "", "b"]),
         ["a", "b"])
```

```gleam
/// Seleciona os valores de *lst*
/// para os quais *pred* devolve True.
fn filtra(
  lst: List(a),
  pred: fn(a) -> Bool,
) -> List(a) {
  case lst {
    [] -> []
    [p, ..r] -> case pred(p) {
      True -> [p, ..filtra(r, pred)]
      False -> filtra(r, pred)
    }
  }
}

check.eq(filtra([3, 2, 7], int.is_even), [2])
check.eq(filtra(["a", "", "b"], eh_nao_vazia),
         ["a", "b"])
```

Exemplo: lista_pares e lista_nao_vazia

```gleam
/// Seleciona os valores pares de *lst*.
fn lista_pares(lst: List(Int)) -> List(Int) {
  filtra(lst, int.is_even)
}

check.eq(lista_pares([3, 2, 7]), [2])
/// Seleciona as strings não vazias de *lst*.
fn lista_nao_vazia(lst: List(String))
                   -> List(String) {
  filtra(lst, eh_nao_vazia)
}

check.eq(lista_nao_vazia(["a", "", "b"]),
         ["a", "b"])
```

```gleam
/// Seleciona os valores de *lst*
/// para os quais *pred* devolve True.
fn filtra(
  lst: List(a),
  pred: fn(a) -> Bool,
) -> List(a) {
  case lst {
    [] -> []
    [p, ..r] -> case pred(p) {
      True -> [p, ..filtra(r, pred)]
      False -> filtra(r, pred)
    }
  }
}

check.eq(filtra([3, 2, 7], int.is_even), [2])
check.eq(filtra(["a", "", "b"], eh_nao_vazia),
         ["a", "b"])
```

filter

filter

Como resultado do exemplo anterior obtivemos a função `filtra`, que é pré-definida em Gleam como `list.filter`.

```
fn comeca_a(s: String) -> Bool {
  string.first(s) == Ok("a")
}


> list.filter(["ana", "pedro", "agua"],
              comeca_a)
["ana", "agua"]
```

```
fn tamanho_1(lst: List(a)) -> Bool {
  case lst {
    [_] -> True
    _ -> False
  }
}


> list.filter([[0], [2, 6], [], [3]],
              tamanho_1)
[[0], [3]]
```

Exemplo: `soma` e `junta_r`

Vamos criar uma função que abstrai o comportamento das funções `soma` e `junta_r`.

Exemplo: soma e junta_r

```gleam
/// Somas os elementos de *lst*.
fn some(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [p, ..r] -> p + soma(r)
  }
}

check.eq(soma([4, 1, 2]), 7)

/// Junta os itens de *lst* em ordem contrária.
fn junta_r(lst: List(String)) -> String {
  case lst {
    [] -> ""
    [p, ..r] -> junta_r(r) <> p
  }
}

check.eq(junta_r(["a", "", "c"]), "ca")
```

Exemplo: soma e junta_r

```gleam
/// Somas os elementos de *lst*.
fn some(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [p, ..r] -> int.add(soma(r), p)
  }
}

check.eq(soma([4, 1, 2]), 7)

/// Junta os itens de *lst* em ordem contrária.
fn junta_r(lst: List(String)) -> String {
  case lst {
    [] -> ""
    [p, ..r] -> string.append(junta_r(r), p)
  }
}

check.eq(junta_r(["a", "", "c"]), "ca")
```

```gleam
fn reduz(lst, init, f) {
  case lst {
    [] -> init
    [p, ..r] -> f(reduz(r, init, f), p)
  }
}
```

Exemplo: soma e junta_r

```
/// Somas os elementos de *lst*.
fn some(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [p, ..r] -> int.add(soma(r), p)
  }
}

check.eq(soma([4, 1, 2]), 7)

/// Junta os itens de *lst* em ordem contrária.
fn junta_r(lst: List(String)) -> String {
  case lst {
    [] -> ""
    [p, ..r] -> string.append(junta_r(r), p)
  }
}

check.eq(junta_r(["a", "", "c"]), "ca")
```

```
fn reduz(lst, init, f) {
  case lst {
    [] -> init
    [p, ..r] -> f(reduz(r, init, f), p)
  }
}
check.eq(
  reduz([4, 1, 2], 0, int.add),
  7)
check.eq(
  reduz(["a", "", "c"], "", string.append),
  "ca")
```

Exemplo: soma e junta_r

```gleam
/// Somas os elementos de *lst*.
fn some(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [p, ..r] -> int.add(soma(r), p)
  }
}

check.eq(soma([4, 1, 2]), 7)

/// Junta os itens de *lst* em ordem contrária.
fn junta_r(lst: List(String)) -> String {
  case lst {
    [] -> ""
    [p, ..r] -> string.append(junta_r(r), p)
  }
}

check.eq(junta_r(["a", "", "c"]), "ca")
```

```gleam
// Reduz os elementos de *lst* em um acumulador
// usando a função *f*. O acumulador começa com *init*
// e é atualizado chamando *f(acc, e)* para cada
// elemento *e* de *lst* da direita para esquerda.
fn reduz(lst, init, f) {
  case lst {
    [] -> init
    [p, ..r] -> f(reduz(r, init, f), p)
  }
}
check.eq(
  reduz([4, 1, 2], 0, int.add),
  7)
check.eq(
  reduz(["a", "", "c"], "", string.append),
  "ca")
```

Exemplo: soma e junta_r

```gleam
/// Somas os elementos de *lst*.
fn some(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [p, ..r] -> int.add(soma(r), p)
  }
}

check.eq(soma([4, 1, 2]), 7)

/// Junta os itens de *lst* em ordem contrária.
fn junta_r(lst: List(String)) -> String {
  case lst {
    [] -> ""
    [p, ..r] -> string.append(junta_r(r), p)
  }
}

check.eq(junta_r(["a", "", "c"]), "ca")
```

```gleam
// Reduz os elementos de *lst* em um acumulador
// usando a função *f*. O acumulador começa com *init*
// e é atualizado chamando *f(acc, e)* para cada
// elemento *e* de *lst* da direita para esquerda.
fn reduz(
  lst: List(a),
  init: b,
  f: fn(b, a) -> b,
) -> b {
  case lst {
    [] -> init
    [p, ..r] -> f(reduz(r, init, f), p)
  }
}
check.eq(
  reduz([4, 1, 2], 0, int.add),
  7)
check.eq(
  reduz(["a", "", "c"], "", string.append),
  "ca")
```

Exemplo: soma e junta_r

```gleam
/// Somas os elementos de *lst*.
fn some(lst: List(Int)) -> Int {
  reduz(lst, 0, int.add)
}

check.eq(soma([4, 1, 2]), 7)

/// Junta os itens de *lst* em ordem contrária.
fn junta_r(lst: List(String)) -> String {
  reduz(lst, "", string.append)
}

check.eq(junta_r(["a", "", "c"]), "ca")
```

```gleam
// Reduz os elementos de *lst* em um acumulador
// usando a função *f*. O acumulador começa com *init*
// e é atualizado chamando *f(acc, e)* para cada
// elemento *e* de *lst* da direita para esquerda.
fn reduz(
  lst: List(a),
  init: b,
  f: fn(b, a) -> b,
) -> b {
  case lst {
    [] -> init
    [p, ..r] -> f(reduz(r, init, f), p)
  }
}
check.eq(
  reduz([4, 1, 2], 0, int.add),
  7)
check.eq(
  reduz(["a", "", "c"], "", string.append),
  "ca")
```

fold_right

fold_right

Como resultado do exemplo anterior obtivemos a função `reduz`, que é pré-definida em Gleam como `list.fold_right`.

```
> list.fold_right([4, 5, 2],
                  1, int.multiply)
40
```

```
> list.fold_right([4, 1, 8], 0, int.max)
8
```

```
fn soma_1_acc(acc: Int, _: Int) -> Int {
  acc + 1
}
```

```
> list.fold_right([4, 1, 8], 0, soma_1_acc)
3
```

Funções `map`, `filter` e `fold_right`

Quando utilizar as funções `map`, `filter` e `fold_right`?

- Quando a lista sempre é processa por inteiro.
- `map`: quando queremos aplicar uma função a cada elemento de uma lista de forma independente.
- `filter`: quando queremos selecionar os elementos de uma lista de acordo com um predicado.
- `fold_right`: quando queremos calcular um resultado de forma incremental analisando cada elemento de uma lista.

Na dúvida, faça o projeto da função recursiva e depois verifique se ela é um caso específico de `map`, `filter` ou `fold_right`.

Exemplo: sinal

Projete uma função que receba como parâmetro uma lista de números e produza uma nova lista com o sinal (1, 0 ou -1) de cada número da lista.

Exemplo: sinal

```gleam
/// Produz uma lista com o sinal de cada
/// elemento de *lst*. O sinal é 1 para
/// positivos, -1 para negativos e 0
/// para neutros.
fn sinais(lst: List(Int)) -> List(Int) {
  todo
}

check.eq(
  sinais([10, 0, 2, -4, -1, 0, 8]),
  [1, 0, 1, -1, -1, 0, 1],
)
```

Podemos usar `map`, `filter` ou `fold_right` para implementar a função? Sim, podemos usar o `map`.

```gleam
fn sinal(n: Int) -> Sinal {
  case n {
    _ if n > 0 -> 1
    _ if n == 0 -> 0
    _ -> -1
  }
}

fn sinais(lst: List(Int)) -> List(Int) {
  list.map(lst, sinal)
}
```

Exemplo: pontos nos eixos

Projete uma função que receba como entrada uma lista de pontos no plano cartesiano e indique quais estão sobre o eixo x ou eixo y.

```
/// Representa um ponto no plano cartesiano.
pub type Ponto {
  Ponto(x: Int, y: Int)
}
```

Exemplo: pontos nos eixos

```gleam
/// Cria uma lista com os elementos de
/// *pontos* que estão sobre o eixo x ou y.
fn seleciona_no_eixo(
  pontos: List(Ponto)
) -> List(Ponto) {
  todo
}
check.eq(
  seleciona_no_eixo([
    Ponto(3, 0), Ponto(1, 3),
    Ponto(0, 2), Ponto(0, 0),
  ]),
  [Ponto(3, 0), Ponto(0, 2), Ponto(0, 0)])
```

Podemos usar `map`, `filter` ou `fold_right` para implementar a função? Sim, podemos usar o `filter`.

```gleam
fn no_eixo(p: Ponto) -> Bool {
  p.x == 0 || p.y == 0
}


fn seleciona_no_eixo(
  pontos: List(Ponto)
) -> List(Ponto) {
  list.filter(pontos, no_eixo)
}
```

Exemplo: ordenação

Projete uma função que receba como entrada uma lista de números e devolva uma lista com os mesmos valores de entrada mas em ordem não decrescente.

Exemplo: ordenação

```
/// Cria uma lista com os elementos de
/// *lst* em ordem não decrescente.
fn ordena(lst: List(Int)) -> List(Int) {
  todo
}

check.eq(ordena([5, -2, 3]), [-2, 3, 5])
```

Podemos usar `map`, `filter` ou `fold_right` para implementar a função? Não está claro... Vamos fazer a implementação usando o modelo.

```
fn ordena(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [p, ..r] -> insere_ordenado(ordena(r), p))
  }
}
```

E então, podemos usar `map`, `filter` ou `fold_right` para implementar a função?

```
fn fold_right(lst, init, f) {
  case lst {
    [] -> []
    [p, ..r] -> f(fold_right(r, init, f), p))
  }
}
```

Sim! Podemos usar o `fold_right`.

```
fn ordena(lst: List(Int)) -> List(Int) {
  list.fold_right(lst, [], insere_ordenado)
}
```

Exercício: projete a função `insere_ordenado`.

Exemplos: maiores strings

Projete uma função que receba como entrada uma lista de strings e devolva uma lista com as strings de tamanho máximo entre todas as strings da lista.

Exemplos: maiores strings

```gleam
/// Cria uma lista com as strings de *lst* que têm tamanho máximo entre todos
/// as strings de *lst*.
fn maiores_strings(lst: List(String)) -> List(String) {
  todo
}

check.eq(
  maiores_strings(["oi", "casa", "aba", "boi", "eita", "a", "cadê"]),
  ["casa", "eita", "cadê"]
)
```

Podemos usar `map`, `filter` ou `fold_right` para implementar a função? Parece complicado...

Vamos separar a solução em duas etapas: encontrar o tamanho máximo e depois selecionar as strings com tamanho máximo.

Exemplos: maiores strings

```gleam
/// Devolve o tamanho máximo entre
/// todos os elementos de *lst*.
fn tamanho_max(lst: List(String) -> Int {
  todo
}

check.eq(
  tamanho_maximo(
    ["oi", "casa", "aba", "boi",
     "eita", "a", "cadê"]),
  4,
)
```

Podemos usar `map`, `filter` ou `fold_right` para implementar a função? Sim, usando o `fold_right`, mas parece complicado...

A função para o `fold_right` teria que fazer duas coisas, determinar o tamanho de uma string e indicar qual é o máximo entre dois tamanhos. Podemos fazer em duas etapas: usamos o `map` para obter uma lista com os tamanhos e o `fold_right` para determinar o valor máximo.

```gleam
/// Devolve o tamanho máximo entre
/// todos os elementos de *lst*.
fn tamanho_max(lst: List(String) -> Int {
  list.fold_rigth(
    list.map(lst, string.length),
    0,
    int.max
  )
}
```

Exemplos: maiores strings

```
fn maiores_string(lst: List(String)) {
  let max = tamanho_maximo(lst)
  // Como definimos a função
  // tem_tamanho_maximo?
  list.filter(lst, tem_tamanho_maximo)
}
```

```
fn maiores_string(lst: List(String)) {
  let max = tamanho_maximo(lst)

  fn tem_tamanho_max(s: String) -> Bool {
    string.length(s) == max
  }

  list.filter(lst, tem_tamanho_maximo)
}
```

O que a função `tem_tamanho_max` tem de diferente?

- É declarada dentro de outra função;
- Acessa uma variável (`max`) que não é um parâmetro, não é global e nem foi definida internamente na função.

Veremos a seguir que este tipo de função tem que ser tratada de forma diferente pelo compilador.

Em Gleam, especificamente, a forma de definir funções desse tipo também é diferente. Por hora, vamos supor que a definição dessa maneira está correta.

Definições locais e fechamentos

Uma **definição local** é aquela que não é feita no escopo global.

Uma **variável livre** em relação a uma função é aquela que não é global, não é um parâmetro da função e nem foi declarada localmente dentro da função.

Como uma função acessa um parâmetro ou uma variáveis local?

Geralmente, consultando o registro de ativação, o quadro, da sua chamada.

Como uma função acessa uma variável livre?

```
fn maiores_string(lst: List(String)) {
  let max = tamanho_maximo(lst)

  fn tem_tamanho_max(s: String) -> Bool {
    string.length(s) == max
  }

  list.filter(lst, tem_tamanho_maximo)
}
```

A variável `max` existe independe da função `tem_tamanho_maximo` estar ativa (executando) ou não, logo ela não pode ser armazenada no registro de ativação de `tem_tamanho_maximo`.

Então, como a variável livre `max` é acessada na função `tem_tamanho_maximo`?

A função `tem_tamanho_maximo` deve “levar” junto com ela a variável livre `max`.

O **ambiente léxico** é uma tabela com referências para as variáveis livres de uma função.

Um **fechamento** (*closure* em inglês) é uma função junto com o seu ambiente léxico.

```
fn maiores_string(lst: List(String)) -> List(String) {
  let max = tamanho_maximo(lst)
  fn tem_tamanho_max(s: String) -> Bool {
    string.length(s) == max
  }
  list.filter(lst, tem_tamanho_maximo)
}
```

Quando a função `tem_tamanho_maximo` é utilizada na chamada de `list.filter` um fechamento é passado como parâmetro.

Exemplo em python

```python
def maiores_strings(lst: list[str]) -> list[str]:
    tmax = tamanho_max(lst)

    def tem_tamanho_max(s: str) -> bool:
        return len(s) == tmax

    return list(filter(tem_tamanho_max, lst))


def tamanho_max(lst: list[str]) -> int:
    # max recebe um iterador
    return max(map(len, lst))
```

Funções anônimas

Quando definimos uma função, estamos especificando duas coisas: **a função** e **o nome da função**.

Da mesma forma que podemos utilizar expressões aritméticas sem precisar nomeá-las, também podemos utilizar funções (de maneira geral, expressões que resultam em funções) sem precisar nomeá-las.

Uma função que não é nomeada é chamada de **função anônima**.

Funções anônimas

```
> // Função anônima
> fn(x: Int) -> Int { x + 1 }
//fn(a) { ... }

> // Chamada de função anônima
> fn(x: Int) -> Int { x + 1 }(3)
4
```

```
> // Armazenando função em variável
> let soma1 = fn(x: Int) -> Int { x + 1 }
//fn(a) { ... }

> // Chamando a função armazenada
> soma1(3)
4
```

Revisão maiores strings

Por questões de simplicidade de projeto, em Gleam, apenas funções anônimas podem ser declaradas dentro de outras funções.

```
fn maiores_string(lst: List(String)) {
  let max = tamanho_maximo(lst)

  // Declaração inválida
  fn tem_tamanho_max(s: String) -> Bool {
    string.length(s) == max
  }

  list.filter(lst, tem_tamanho_maximo)
}
```

```
fn maiores_string(lst: List(String)) {
  let max = tamanho_maximo(lst)
  let tem_tamanho_max = fn(s: String) -> Bool {
    string.length(s) == max
  }
  list.filter(lst, tem_tamanho_maximo)
}
```

Ou sem armazenar a função em uma variável:

```
fn maiores_string(lst: List(String)) {
  let max = tamanho_maximo(lst)
  list.filter(fn(s) { string.length(s) == max })
}
```

Em que situações devemos utilizar um funções anônimas?

Como parâmetro, quando a função for pequena e necessária apenas naquele local:

```
> list.map([3, 8, -6], fn(x) { x * 2 })
[6, 16, -12]
> list.filter([3, 20, -4, 48], fn(x) { x < 10 })
[3, -4]
```

Em Python

```
>>> list(map(lambda x: x * 2, [3, 8, -6]))
[6, 16, -12]
>>> list(filter(lambda x: x < 10, [3, 20, -4, 48])
[3, -4]
```

Defina a função `mapeia` em termos da função `reduz`.

```
fn mapeia(lst, f) {
  reduz(lst, fn(acc, e) {
    [f(e), ..acc]
  })
}
```

Defina a função `filtra` em termos da função `reduz`.

```
fn filtra(lst, pred) {
  reduz(lst, fn(acc, e) {
    case pred(e) {
      True -> [e, ..acc]
      False -> acc
    }
  })
}
```

Em que situações devemos utilizar um funções anônimas?

- Como parâmetro, quando a função for pequena e necessária apenas naquele local.
- Como resultado de funções.

Funções que produzem funções

Como identificar a necessidade de criar e utilizar funções que produzem funções?

- Parametrizar a criação de funções fixando alguns parâmetros;
- Composição de funções;
- ...

Exemplo: somador

Defina uma função que receba um parâmetro *n* e devolva uma função que soma o seu argumento a *n*.

Exemplo: somador

```
> let soma1 = soma(1)
//fn(a) { ... }
> soma1(4)
5
> soma1(6)
7

> soma(1)(2)
3

> list.map([4, 1, 3], soma(5))
[9, 6, 8]
```

```
/// Devolve uma função que recebe um
/// parâmetro *x* e faz a soma de *n* e *x*.
pub fn soma(n: Int) -> fn(Int) -> Int {
  todo
}

pub fn soma(n: Int) -> fn(Int) -> Int {
  fn(x: Int) -> Int { n + x }
}

pub fn soma(n: Int) -> fn(Int) -> Int {
  fn(x) { n + x }
}
```

Exemplo: negação

Defina uma função que receba como parâmetro um predicado (função de um parâmetro que produz booleano) e devolve uma função que devolve a negação do predicado.

Exemplo: negação

```
> nega(list.is_empty)([])
False
> nega(list.is_empty)([1, 2])
True

> filter([4, 1, 2, 0, 3], nega(int.is_odd))
[4, 2, 0]
```

```
/// Devolve uma função que é semelhante a
/// *pred*, mas que devolve a negação do
/// resultado de *pred*.
pub fn nega(
  pred: fn(a) -> Bool
) -> fn(a) -> Bool {
  todo
}

pub fn nega(
  pred: fn(a) -> Bool
) -> fn(a) -> Bool {
  fn(x) { !pred(x) }
}
```

Açúcar sintático

Açúcar sintático

Um **açúcar sintático** (*syntatic sugar*) é uma construção sintática de uma linguagens de programação que deixam o seu uso mais simples, ou doce, para os humanos.

Vamos ver alguns açucares sintáticos do Gleam.

O uso de fechamentos com um parâmetro é bastante comum, por isso, o Gleam oferece uma forma abreviada para criá-los.

Um fechamento da forma `fn(x) { f(..., x, ...) }`, onde `f` é uma função qualquer e `...` são as variáveis livres do fechamento, pode ser escrito de forma abreviada como `f(..., _, ...)`, onde o marcador de posição `_` define o parâmetro para o fechamento.

```
f(..., _, ...)
```

é a mesma coisa que

```
fn(x) { f(..., x, ...) }
```

Fechamento abreviado

```
> // separa a string em ","
> let sep = ","
> let separa = fn(s) { string.split(s, sep) }
// fn(a) { ... }
> separa("12,2,-1")
["12", "2", "-1"]
```

```
> // forma abreviada
> let sep = ","
> let separa = string.split(_, sep)
// fn(a) { ... }
> separa("12,2,-1")
["12", "2", "-1"]
```

```
> // seleciona os elementos de a que estão em b
> let a = [1, 4, 2]
> let b = [3, 2, 7, 1]
> list.filter(a, fn(e) { list.contains(b, e) })
[1, 2]
```

```
> // de forma abreviada
> let a = [1, 4, 2]
> let b = [3, 2, 7, 1]
> list.filter(a, list.contains(b, _))
[1, 2]
```

```
> // soma 1 em cada elemento da lista
> list.map([3, 1, 4], fn(x) { x + 1 })
[4, 2, 5]
> // em uma forma que pode ser abreviada
> list.map([3, 1, 4], fn(x) { int.add(x, 1) })
[4, 2, 5]
```

```
> // de forma abreviada
> list.map([3, 1, 4], int.add(_, 1))
[4, 2, 5]
```

Um **cadeia de processamento**, ou *pipeline*, é uma sequência de operações onde a saída de uma operação é utilizada como entrada da próxima.

A forma "comum" de chamar funções pode não ser adequada para uma cadeia de processamento com muitas etapas, isso porque a ordem de execução fica de "dentro para fora", o que dificulta a escrita e leitura do código.

Gleam oferece o operador binário `|>` para facilitar as cadeias de processamento

```
a |> b(x, ..., z)
```

é equivalente a

```
b(a, x, ..., z) ou b(x, ..., z)(a)
```

Cadeia de processamento

```
fn tamanho_max(lst: List(String) -> Int {
  list.fold_rigth(list.map(lst, string.length), 0, int.max)
}

fn tamanho_max(lst: List(String) -> Int {
  list.fold_rigth(
    list.map(lst, string.length),
    0,
    int.max,
  )
}

fn tamanho_max(lst: List(String) -> Int {
  lst
  |> list.map(string.length)
  |> list.fold_right(0, int.max)
}
```

```
// cria uma lista com todos os nomes que começam com a letra "a"
// enumerados em ordem alfabética.
> enumera_em_ordem_comeca_a(["pedro", "angela", "joao", "ana", "aline"])
["1. aline", "2. ana", "3. angela"]

fn enumera_em_ordem_comeca_a(nomes: List(String)) -> List(String) {
  list.index_map(
    list.sort(list.filter(nomes, string.starts_with(_, "a")), string.compare),
    fn (nome, num) { int.to_string(num) <> ". " <> nome },
 )
}

fn enumera_em_ordem_comeca_a(nomes: List(String)) -> List(String) {
  nomes
  |> list.filter(string.starts_with(_, "a"))
  |> list.sort(string.compare)
  |> list.index_map(fn (nome, num) { int.to_string(num) <> ". " <> nome })
}
```

As funções de alta ordem e o casamento de padrão são essenciais para a programação funcional. No entanto, em algumas situações, o uso dessas construções pode gerar indentação excessiva.

Vamos rever um exemplo que vimos em tipos de dados.

Use

```gleam
fn soma(a: String, b: String)
  -> Result(Int, Nil) {
  case int.parse(a) {
    Ok(x) -> case int.parse(b) {
      Ok(y) -> Ok(x + y)
      Error(err) -> Error(err)
    }
    Error(err) -> Error(err)
  }
}
```

Podemos melhorar?

Existe um padrão recorrente no código: se o resultado for `Ok`, então, continue com outra operação, senão, pare e devolva o erro.

Podemos criar uma função de alta ordem `entao` para abstrair esse padrão.

```gleam
/// Excuta *fun* com o valor de *r* e
/// devolve seu resultado se *r* é Ok,
/// senão devolve o mesmo erro de *r*.
> entao(Ok(10), fn(x) { Ok(int.to_string(x)) })
Ok("10")
> entao(Error("a"), fn(x) { Ok(int.to_string(x)) })
Error("falhou")
> Ok("casa") |> entao(string.first)
Ok("c")

fn entao(
  r: Result(a, b),
  fun: fn(a) -> Result(c, b),
) -> Result(c, b) {
  case r {
    Ok(value) -> fun(value)
    Error(error) -> Error(error)
  }
}
```

```gleam
fn soma(a: String, b: String)
  -> Result(Int, Nil) {
  case int.parse(a) {
    Ok(x) -> case int.parse(b) {
      Ok(y) -> Ok(x + y)
      Error(err) -> Error(err)
    }
    Error(err) -> Error(err)
  }
}
```

Podemos melhorar?

Existe um padrão recorrente no código: se o resultado for Ok, então, continue com outra operação, senão, pare e devolva o erro.

Podemos criar uma função de alta ordem `entao` para abstrair esse padrão.

```gleam
fn soma(a: String, b: String) -> Result(Int, Nil) {
  entao(int.parse(a), fn(x) {
    entao(int.parse(b), fn(y) {
      Ok(x + y)
    })
  })
}
```

A função `entao` está definida na biblioteca padrão como `result.then` e `result.try` (as duas funções fazem a mesma coisa).

```gleam
fn soma(a: String, b: String) -> Result(Int, Nil) {
  result.try(int.parse(a), fn(x) {
    result.try(int.parse(b), fn(y) {
      Ok(x + y)
    })
  })
}
```

Use

```gleam
fn soma(a: String, b: String)
  -> Result(Int, Nil) {
  case int.parse(a) {
    Ok(x) -> case int.parse(b) {
      Ok(y) -> Ok(x + y)
      Error(err) -> Error(err)
    }
    Error(err) -> Error(err)
  }
}
```

```gleam
fn soma(a: String, b: String)
  -> Result(Int, Nil) {
  result.try(int.parse(a), fn(x) {
    result.try(int.parse(b), fn(y) {
      Ok(x + y)
    })
  })
}
```

Podemos melhorar? Sim!

```gleam
fn soma(a: String, b: String)
  -> Result(Int, Nil) {
  use x <- result.try(int.parse(a))
  use y <- result.try(int.parse(b))
  Ok(x + y)
}
```

Apesar de parecem diferentes, as duas últimas definições da função `soma` são equivalentes!
O **`use`** é apenas açúcar sintático para definir uma função anônima e passá-la como último parâmetro para uma chamada de função.

Use

O **use** permite utilizar funções anônimas como parâmetros sem aumentar a indentação do código.
Uma chamada da forma

```
funcao(a, b, ..., fn(x, y, ...) { corpo })
```

pode ser escrita como

```
use x, y, ... <- funcao(a, b, ...)
corpo
```

A função que está sendo chamada pode receber qualquer quantidade de parâmetros, mas o último parâmetro precisa ser uma função.
A função que está sendo passada como parâmetro também pode receber qualquer quantidade de parâmetros.
No **use** os parâmetros para a função anônima ficam do lado esquerdo de **<-** e a função que está sendo chamada fica do lado direito. O corpo da função anônima inclui todo o código que está após o **use**, até fechar o bloco atual.

Outras funções de alta ordem

Outras funções de alta ordem

```
bool
  guard
  lazy_guard
```

```
option
  lazy_or
  lazy_unwrap
  map
  or
  then
```

```
result
  lazy_or
  lazy_unwrap
  map
  map_error
  then
  try
  try_recover
```

```
list
  all
  any
  drop_while
  filter_map
  find_map
  fold_until
  index_fold
  index_map
  map2
  map_fold
  sort
  split_while
  take_while
  try_fold
  try_map
```

```
dict
  filter
  fold
  map_values
```

```
set
  filter
  fold
  map
```

Referências

Referências

Básicas

- Vídeos Abstractions
- Capítulo 15 do livro HTDP
- Fechamento abreviado, pipelines, use e use sugar do tour do Gleam.

Complementares

- Seções 1.3 (1.3.1 e 1.3.2) e 2.2.3 do livro SICP
- Seções 4.2 e 5.5 do livro TSPL4