

Autorreferência e recursividade

Parte I

Programação Funcional

Marco A L Barbosa

malbarbo.pro.br

Departamento de Informática

Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhagual 4.0 Internacional.

<http://github.com/malbarbo/na-progfun>

Introdução

Projete uma função que some uma sequência de números.

Como representar e processar uma quantidade de dados arbitrária?

- Vamos criar tipos de dados com autorreferência
- Vamos usar funções recursivas para processar dados com autorreferência

Um **tipo de dado com autorreferência** é aquele definido em termos de si mesmo, de forma direta ou indireta.

Uma **função recursiva** é aquela que chama a si mesma, de forma direta ou indireta.

Listas

O tipo de dado com autorreferência mais comum nas linguagens funcionais é a lista.

Vamos tentar criar uma definição para lista de números.

A ideia é criar uma estrutura com dois campos. O primeiro campo representa o primeiro item na lista e o segundo campo representa o restante da lista (que é uma lista).

```
type Lista {  
  Lista(primeiro: Int, resto: Lista)  
}
```

Observe a autorreferência!

Utilizando esta definição, vamos tentar criar uma lista com os valores 4, 2 e 8.

```
let lst = Lista(4, Lista(2, Lista(8, ...)))
```

O problema com esta definição é que as listas não tem fim. Uma lista tem uma parte que é uma lista, que tem uma parte que é uma lista, etc. Ou seja, a definição não é bem formada.

Para ser **bem formada**, uma definição com autorreferência deve ter:

- Pelo menos um caso base (sem autorreferência)
- Pelo menos um caso com autorreferência

Os casos base descrevem valores que podem ser criados diretamente.

Os casos com autorreferência permitem a criação de novos valores a partir de valores existentes.

O que está faltando na nossa definição de lista? Um caso base, ou seja, uma forma de criar uma lista que não dependa de outra lista. Que lista pode ser essa?

A lista vazia.

Uma **lista** é:

- Vazia;
- Ou não vazia, contendo o primeiro e o resto, que é uma **lista**.

Em Gleam

```
type Lista {  
  Vazia  
  NaoVazia(primeiro: Int, resto: Lista)  
}
```

```
> // lista vazia
> let lst0: Lista = Vazia
Vazia

> // lista com o 3
> let lst1: Lista = NaoVazia(3, Vazia)
NaoVazia(primeiro: 3, resto: Vazia)

> // Lista com o 8 e 7
> let lst2: Lista = NaoVazia(8, NaoVazia(7, Vazia))
NaoVazia(primeiro: 8, resto: NaoVazia(primeiro: 7, resto: Vazia))

> // Nova lista com o 3 como primeiro, seguido de lst2
> NaoVazia(3, lst2)
NaoVazia(primeiro: 3, resto: NaoVazia(primeiro: 8, resto: NaoVazia(primeiro: 7, resto: Vazia)))
```

Como consultar o primeiro elemento de uma lista?

```
> lst1.primeiro
```

```
error: Unknown record field
```

```
1 | lst1.primeiro
  |      ^^^^^^^ This field does not exist
```

The value being accessed has this type:

Lista

It does not have any fields.

Note: The field you are trying to access might not be consistently present or positioned across the custom type's variants, preventing reliable access. Ensure the field exists in the same position and has the same type in all variants to enable direct accessor syntax.

```
> case lst1 {  
  Vazia -> todo  
  NaoVazia(primeiro, _) -> primeiro  
}  
3
```

Nós vimos anteriormente que o tipo de dado de entrada de uma função sugere uma forma para o corpo da função.

- Qual é a forma do corpo da função que um tipo enumerado de entrada sugere? Um **case** com um caso para cada valor da enumeração.
- Qual é a forma do corpo da função que um tipo união de entrada sugere? Um **case** com um caso para cada classe da união.

Qual é a forma do corpo da função que o tipo de entrada **Lista** sugere?

Uma condicional com dois casos:

- A lista é vazia
- A lista não é vazia

Em Gleam

```
pub fn fn_para_lista(lst: Lista) {  
  case lst {  
    Vazia -> todo  
    NaoVazia(primeiro, resto) -> todo  
  }  
}
```

Qual é o tipo de **primeiro**? Um inteiro, que é um valor atômico.

Qual é o tipo de **resto**? Uma lista, que é uma união.

Um valor atômico pode ser processado diretamente, mas como processar uma lista? Fazendo análise dos casos...

Vamos fazer uma alteração no modelo `fn_para_lista` e adicionar uma chamada recursiva para processar **resto**. Essa alteração pode parecer meio “mágica” agora, mas ela vai ficar mais clara em breve.

```
type Lista {  
  Vazia  
  NaoVazia(primeiro: Int, resto: Lista)  
}
```

Modelo para função para listas

```
pub fn fn_para_lista(lst: Lista) {  
  case lst {  
    Vazia -> todo  
    NaoVazia(primeiro, resto) -> {  
      todo  
      primeiro  
      fn_para_lista(resto)  
    }  
  }  
}
```

Quais são as relações entre a definição de **Lista** e **fn_para_lista**?

- A definição tem dois casos, o modelo também;
- Na definição o **resto** é um **autorreferência**, na função a **recursão** é feita como o **resto**.

Defina uma função que some os valores de uma lista de números.

Exemplo: soma - especificação

```
/// Soma os valores de *lst*  
pub fn soma(lst: Lista) -> Int {  
  todo  
}
```

```
pub fn soma_examples() {  
  check.eq(soma(Vazia), 0)  
  check.eq(soma(NaoVazia(3, Vazia)), 3)  
  check.eq(  
    soma(NaoVazia(5, NaoVazia(3, Vazia))),  
    8,  
  )  
  check.eq(  
    soma(NaoVazia(2, NaoVazia(5, NaoVazia(3, Vazia)))),  
    10,  
  )  
}
```

E agora, como escrevemos a implementação? Vamos partir do modelo de função para listas.

Exemplo: soma - implementação

```
/// Soma os valores de *lst*
pub fn soma(lst: Lista) -> Int {
  case lst {
    Vazia -> todo
    NaoVazia(primeiro, resto) -> {
      todo
      primeiro
      soma(resto)
    }
  }
}
```

```
pub fn soma_examples() {
  check.eq(soma(Vazia), 0)
  check.eq(soma(NaoVazia(3, Vazia)), 3)
  check.eq(
    soma(NaoVazia(5, NaoVazia(3, Vazia))),
    8,
  )
  check.eq(
    soma(NaoVazia(2, NaoVazia(5, NaoVazia(3, Vazia)))),
    10,
  )
}
```

Agora precisamos preencher as lacunas. Qual deve ser o resultado quando a lista é vazia? 0.

Exemplo: soma - implementação

```
/// Soma os valores de *lst*
pub fn soma(lst: Lista) -> Int {
  case lst {
    Vazia -> 0
    NaoVazia(primeiro, resto) -> {
      todo
      primeiro
      soma(resto)
    }
  }
}
```

```
pub fn soma_examples() {
  check.eq(soma(Vazia), 0)
  check.eq(soma(NaoVazia(3, Vazia)), 3)
  check.eq(
    soma(NaoVazia(5, NaoVazia(3, Vazia))),
    8,
  )
  check.eq(
    soma(NaoVazia(2, NaoVazia(5, NaoVazia(3, Vazia)))),
    10,
  )
}
```

Agora precisamos analisar o caso em que a lista não é vazia. O modelo está sugerindo fazer uma chamada recursiva para o resto da lista. Aqui vem o ponto crucial!

Exemplo: soma - implementação

```
/// Soma os valores de *lst*
pub fn soma(lst: Lista) -> Int {
  case lst {
    Vazia -> 0
    NaoVazia(primeiro, resto) -> {
      todo
      primeiro
      soma(resto)
    }
  }
}
```

```
pub fn soma_examples() {
  check.eq(soma(Vazia), 0)
  check.eq(soma(NaoVazia(3, Vazia)), 3)
  check.eq(
    soma(NaoVazia(5, NaoVazia(3, Vazia))),
    8,
  )
  check.eq(
    soma(NaoVazia(2, NaoVazia(5, NaoVazia(3, Vazia)))),
    10,
  )
}
```

Mesmo a função não estando completa, nós vamos **assumir** que ela produz a resposta correta para o resto da lista. Tendo a soma do resto e o primeiro, como obtermos a soma da lista? Somando os dois.

Exemplo: soma - implementação

```
/// Soma os valores de *lst*
pub fn soma(lst: Lista) -> Int {
  case lst {
    Vazia -> 0
    NaoVazia(primeiro, resto) ->
      primeiro + soma(resto)
  }
}
```

```
pub fn soma_examples() {
  check.eq(soma(Vazia), 0)
  check.eq(soma(NaoVazia(3, Vazia)), 3)
  check.eq(
    soma(NaoVazia(5, NaoVazia(3, Vazia))),
    8,
  )
  check.eq(
    soma(NaoVazia(2, NaoVazia(5, NaoVazia(3, Vazia)))),
    10,
  )
}
```

Verificação: ok. (Revisão) Podemos melhorar o código?

A linguagem Gleam já fornece o tipo **List** e uma notação amigável para criar e desestruturar listas.

```
> // Lista vazia
> let lst0: List(Int) = []
[]

> // Lista com 3 e 8
> let lst1: List(Int) = [3, 8]
[3, 8]

> // Nova lista a partir de uma existente
> let lst2 = [7, ..lst1]
[7, 3, 8]
```

```
> // Desestruturação
> case lst2 {
  [] -> todo
  [primeiro, ..resto] -> primeiro
}
7

> case lst2 {
  [] -> todo
  [primeiro, ..resto] -> resto
}
[3, 8]
```

List tem a mesma estrutura da lista que definimos, a diferença é apenas na sintaxe!

Uma **Lista** é:

- **Vazia**;
- **NaoVazia**(primeiro, resto), onde resto é uma **Lista**.

```
pub fn fn_para_lista(lst: Lista) {  
  case lst {  
    Vazia -> todo  
    NaoVazia(primeiro, resto) -> {  
      todo  
      primeiro  
      fn_para_lista(resto)  
    }  
  }  
}
```

Uma **List** é:

- `[]`; ou
- `[primeiro, ..resto]`, onde resto é uma **List**.

```
pub fn fn_para_list(lst: List(a)) {  
  case lst {  
    [] -> todo  
    [primeiro, ..resto] -> {  
      todo  
      primeiro  
      fn_para_list(resto)  
    }  
  }  
}
```

Uma **Lista** é:

- **Vazia**;
- **NaoVazia**(primeiro, resto), onde resto é uma **Lista**.

```
pub fn soma(lst: Lista) {  
  case lst {  
    Vazia -> 0  
    NaoVazia(primeiro, resto) ->  
      primeiro + soma(resto)  
  }  
}
```

Uma **List** é:

- `[]`; ou
- `[primeiro, ..resto]`, onde resto é uma **List**.

```
pub fn soma(lst: List(a)) {  
  case lst {  
    [] -> 0  
    [primeiro, ..resto] ->  
      primeiro + soma(resto)  
  }  
}
```


Defina uma função que verifique se um dado valor está em uma lista de números.

Exemplo: contém - especificação

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List<Int>, v: Int) -> Bool {  
    todo  
}  
  
pub fn contem_examples() {  
    check.eq(contem([], 3), False)  
    check.eq(contem([3], 3), True)  
    check.eq(contem([3], 4), False)  
    check.eq(contem([4, 10, 3], 4), True)  
    check.eq(contem([4, 10, 3], 10), True)  
    check.eq(contem([4, 10, 3], 8), False)  
}
```

Como começamos a implementação? Com o modelo.

Exemplo: contém - implementação

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List<Int>, v: Int) -> Bool {  
  case lst {  
    [] -> { todo v }  
    [primeiro, ..resto] -> {  
      todo  
      v  
      primeiro  
      contem(resto, v)  
    }  
  }  
}
```

```
pub fn contem_examples() {  
  check.eq(contem([], 3), False)  
  check.eq(contem([3], 3), True)  
  check.eq(contem([3], 4), False)  
  check.eq(contem([4, 10, 3], 4), True)  
  check.eq(contem([4, 10, 3], 10), True)  
  check.eq(contem([4, 10, 3], 8), False)  
}
```

O esboço para cada caso começa com um **inventário** dos valores disponíveis para implementar aquele caso. Por isso adicionamos `v` em cada caso.

Exemplo: contém - implementação

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List<Int>, v: Int) -> Bool {  
  case lst {  
    [] -> { todo v }  
    [primeiro, ..resto] -> {  
      todo  
      v  
      primeiro  
      contem(resto, v)  
    }  
  }  
}
```

```
pub fn contem_examples() {  
  check.eq(contem([], 3), False)  
  check.eq(contem([3], 3), True)  
  check.eq(contem([3], 4), False)  
  check.eq(contem([4, 10, 3], 4), True)  
  check.eq(contem([4, 10, 3], 10), True)  
  check.eq(contem([4, 10, 3], 8), False)  
}
```

O que fazemos agora? Implementamos o caso base.

Exemplo: contém - implementação

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List<Int>, v: Int) -> Bool {  
  case lst {  
    [] -> False  
    [primeiro, ..resto] -> {  
      todo  
      v  
      primeiro  
      contem(resto, v)  
    }  
  }  
}
```

```
pub fn contem_examples() {  
  check.eq(contem([], 3), False)  
  check.eq(contem([3], 3), True)  
  check.eq(contem([3], 4), False)  
  check.eq(contem([4, 10, 3], 4), True)  
  check.eq(contem([4, 10, 3], 10), True)  
  check.eq(contem([4, 10, 3], 8), False)  
}
```

Assumindo que a função produz a resposta correta para o resto (determina se **v** está no **resto**), com podemos determinar se **v** está **lst**?

Exemplo: contém - implementação

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List<Int>, v: Int) -> Bool {  
  case lst {  
    [] -> False  
    [primeiro, ..resto] ->  
      case v == primeiro {  
        True -> True  
        False -> contem(resto, v)  
      }  
  }  
}
```

```
pub fn contem_examples() {  
  check.eq(contem([], 3), False)  
  check.eq(contem([3], 3), True)  
  check.eq(contem([3], 4), False)  
  check.eq(contem([4, 10, 3], 4), True)  
  check.eq(contem([4, 10, 3], 10), True)  
  check.eq(contem([4, 10, 3], 8), False)  
}
```

Verificação: ok. (Revisão) Podemos melhorar o código?

Exemplo: contém - revisão

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List<Int>, v: Int) -> Bool {  
  case lst {  
    [] -> False  
    [primeiro, ..resto] ->  
      v == primeiro || contem(resto, v)  
  }  
}
```

```
pub fn contem_examples() {  
  check.eq(contem([], 3), False)  
  check.eq(contem([3], 3), True)  
  check.eq(contem([3], 4), False)  
  check.eq(contem([4, 10, 3], 4), True)  
  check.eq(contem([4, 10, 3], 10), True)  
  check.eq(contem([4, 10, 3], 8), False)  
}
```

Exemplo: contém - revisão

```
/// Devolve True se *v* está em *lst*,  
/// False caso contrário.  
pub fn contem(lst: List(a), v: a) -> Bool {  
  case lst {  
    [] -> False  
    [primeiro, ..resto] ->  
      v == primeiro || contem(resto, v)  
  }  
}
```

```
pub fn contem_examples() {  
  check.eq(contem([], 3), False)  
  check.eq(contem([3], 3), True)  
  check.eq(contem([3], 4), False)  
  check.eq(contem([4, 10, 3], 4), True)  
  check.eq(contem(["a", "b"], "b"), True)  
  check.eq(contem(["a", "b"], "c"), False)  
}
```

Estamos utilizando **polimorfismo paramétrico** para deixar a função **contem** genérica.

A função **contem** é genérica em relação ao tipo **a**, que é um parâmetro que pode assumir (implicitamente) qualquer tipo.

Note que o tipo dos elementos da lista deve ser o mesmo que o tipo de **v**.

Defina uma função que soma um valor x em cada elemento de uma lista de números.

Exemplo: soma x - especificação

```
/// Soma *x* a cada elemento de *lst*.
```

```
pub fn soma_x(  
  lst: List(Int),  
  x: Int  
) -> List(Int) {  
  todo  
}
```

```
pub fn soma_x_examples() {  
  check.eq(soma_x([], 4), [])  
  check.eq(soma_x([4, 2], 5), [9, 7])  
  check.eq(soma_x([3, -1, 4], -2), [1, -3, 2])  
}
```

Como começamos a implementação? Com o modelo.

Exemplo: soma x - implementação

```
/// Soma *x* a cada elemento de *lst*.
```

```
pub fn soma_x(lst, x) -> List(Int) {
```

```
  case lst {
```

```
    [] -> { todo x }
```

```
    [primeiro, ..resto] -> {
```

```
      todo
```

```
      x
```

```
      primeiro
```

```
      soma_x(resto, x)
```

```
    }
```

```
  }
```

```
}
```

```
pub fn soma_x_examples() {
```

```
  check.eq(soma_x([], 4), [])
```

```
  check.eq(soma_x([4, 2], 5), [9, 7])
```

```
  check.eq(soma_x([3, -1, 4], -2), [1, -3, 2])
```

```
}
```

O que fazemos agora? Implementamos o caso base.

Exemplo: soma x - implementação

```
/// Soma *x* a cada elemento de *lst*.
pub fn soma_x(lst, x) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] -> {
      todo
      x
      primeiro
      soma_x(resto, x)
    }
  }
}
```

```
pub fn soma_x_examples() {
  check.eq(soma_x([], 4), [])
  check.eq(soma_x([4, 2], 5), [9, 7])
  check.eq(soma_x([3, -1, 4], -2), [1, -3, 2])
}
```

Assumindo que a função produz a resposta correta para o resto (soma x em cada elemento do `resto`), com podemos criar uma lista somando x em cada elemento de `lst`?

Exemplo: soma x - implementação

```
/// Soma *x* a cada elemento de *lst*.
pub fn soma_x(lst, x) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] ->
      [x + primeiro, ..soma_x(resto, x)]
  }
}
```

Verificação: Ok.

```
pub fn soma_x_examples() {
  check.eq(soma_x([], 4), [])
  check.eq(soma_x([4, 2], 5), [9, 7])
  check.eq(soma_x([3, -1, 4], -2), [1, -3, 2])
}
```

Defina uma função que remova todos os números negativos de uma lista de números.

Exemplo: remove negativos - especificação

```
// Cria uma nova lista sem
// os valores negativos de *lst*.
pub fn remove_negativos(
  lst: List(Int)
) -> List(Int) {
  todo
}
```

```
pub fn remove_negativos_examples() {
  check.eq(
    remove_negativos([]),
    [],
  )
  check.eq(
    remove_negativos([-1, 2, -3]),
    [2],
  )
  check.eq(
    remove_negativos([3, 4, -2]),
    [3, 4],
  )
}
```

Como começamos a implementação? Com o modelo.

Exemplo: remove negativos - implementação

```
// Cria uma nova lista sem
// os valores negativos de *lst*.
pub fn remove_negativos(lst) -> List(Int) {
  case lst {
    [] -> todo
    [primeiro, ..resto] -> {
      todo
      primeiro
      remove_negativos(resto)
    }
  }
}
```

```
pub fn remove_negativos_examples() {
  check.eq(
    remove_negativos([]),
    [],
  )
  check.eq(
    remove_negativos([-1, 2, -3]),
    [2],
  )
  check.eq(
    remove_negativos([3, 4, -2]),
    [3, 4],
  )
}
```

O que fazemos agora? Implementamos o caso base.

Exemplo: remove negativos - implementação

```
// Cria uma nova lista sem
// os valores negativos de *lst*.
pub fn remove_negativos(lst) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] -> {
      todo
      primeiro
      remove_negativos(resto)
    }
  }
}
```

```
pub fn remove_negativos_examples() {
  check.eq(
    remove_negativos([]),
    [],
  )
  check.eq(
    remove_negativos([-1, 2, -3]),
    [2],
  )
  check.eq(
    remove_negativos([3, 4, -2]),
    [3, 4],
  )
}
```

Assumindo que a função produz a resposta correta para o resto (remove os negativos de `resto`), com podemos remover os negativos de `lst`?

Exemplo: remove negativos - implementação

```
// Cria uma nova lista sem
// os valores negativos de *lst*.
pub fn remove_negativos(lst) -> List(Int) {
  case lst {
    [] -> todo
    [primeiro, ..resto] ->
      case primeiro < 0 {
        True -> remove_negativos(resto)
        False ->
          [primeiro,
           ..remove_negativos(resto)]
      }
  }
}
```

```
pub fn remove_negativos_examples() {
  check.eq(
    remove_negativos([]),
    [],
  )
  check.eq(
    remove_negativos([-1, 2, -3]),
    [2],
  )
  check.eq(
    remove_negativos([3, 4, -2]),
    [3, 4],
  )
}
```

Verificação: ok. (Revisão) Podemos melhorar o código?

Exemplo: remove negativos - revisão

```
// Cria uma nova lista sem
// os valores negativos de *lst*.
pub fn remove_negativos(lst) -> List(Int) {
  case lst {
    [] -> todo
    [primeiro, ..resto] if primeiro < 0 ->
      remove_negativos(resto)
    [primeiro, ..resto] ->
      [primeiro, ..remove_negativos(resto)]
  }
}
```

```
pub fn remove_negativos_examples() {
  check.eq(
    remove_negativos([]),
    [],
  )
  check.eq(
    remove_negativos([-1, 2, -3]),
    [2],
  )
  check.eq(
    remove_negativos([3, 4, -2]),
    [3, 4],
  )
}
```

Um dicionário é um TAD que associa chaves com valores. Existem diversas formas de implementar um dicionário, a mais simples é utilizando **listas de associações** chave-valor. Apesar dos tempos de inserção e buscar serem lineares, na prática, para poucas chaves, a implementação é adequada.

- a) Defina um tipo de dado que represente uma associação entre uma string e um número.
- b) Projete uma função que determine, a partir de uma lista de associações, qual é o valor associado com uma string.

Exemplo: número de ocorrências - especificação

```
// Associação entre chave e valor.
type Par {
  Par(chave: String, valor: Int)
}

/// Devolve o valor associado com *chave* em *lst* ou Error(Nil) se *s* não
/// aparece como chave em *lst*.
pub fn busca(lst: List(Par), chave: String) -> Result(Int, Nil) {
  todo
}

pub fn busca_examples() {
  check.eq(busca([], "casa"), Error(Nil))
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "casa"), Error(Nil))
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "nada"), Ok(3))
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "outra"), Ok(2))
}
```

Exemplo: número de ocorrências - implementação

```
pub fn busca_examples() {
  check.eq(busca([], "casa"), Error(Nil))
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "casa"), Error(Nil))
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "nada"), Ok(3))
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "outra"), Ok(2))
}

pub fn busca(lst: List(Par), chave: String) -> Result(Int, Nil) {
  case lst {
    [] -> { todo chave }
    [primeiro, ..resto] -> {
      todo chave
        primeiro
        busca(resto, chave)
    }
  }
}
```

Exemplo: número de ocorrências - implementação

```
pub fn busca_examples() {  
  check.eq(busca([], "casa"), Error(Nil))  
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "casa"), Error(Nil))  
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "nada"), Ok(3))  
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "outra"), Ok(2))  
}
```

```
pub fn busca(lst: List(Par), chave: String) -> Result(Int, Nil) {  
  case lst {  
    [] -> Error(Nil)  
    [primeiro, ..resto] -> {  
      todo chave  
      primeiro  
      busca(resto, chave)  
    }  
  }  
}
```

Exemplo: número de ocorrências - implementação

```
pub fn busca_examples() {  
    check.eq(busca([], "casa"), Error(Nil))  
    check.eq(busca([Par("nada", 3), Par("outra", 2)], "casa"), Error(Nil))  
    check.eq(busca([Par("nada", 3), Par("outra", 2)], "nada"), Ok(3))  
    check.eq(busca([Par("nada", 3), Par("outra", 2)], "outra"), Ok(2))  
}
```

```
pub fn busca(lst: List(Par), chave: String) -> Result(Int, Nil) {  
    case lst {  
        [] -> Error(Nil)  
        [primeiro, ..resto] -> {  
            case primeiro.chave == chave {  
                True -> Ok(primeiro.valor)  
                False -> busca(resto, chave)  
            }  
        }  
    }  
}
```


Exemplo: número de ocorrências - revisão

```
pub fn busca_examples() {  
  check.eq(busca([], "casa"), Error(Nil))  
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "casa"), Error(Nil))  
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "nada"), Ok(3))  
  check.eq(busca([Par("nada", 3), Par("outra", 2)], "outra"), Ok(2))  
}  
  
pub fn busca(lst: List(Par), chave: String) -> Result(Int, Nil) {  
  case lst {  
    [] -> Error(Nil)  
    [primeiro, ..] if primeiro.chave == chave -> Ok(primeiro.valor)  
    [_, ..resto] -> busca(resto, chave)  
  }  
}
```

Exemplo: número de ocorrências - revisão

```
type Par(a, b) {  
  Par(chave: a, valor: b)  
}  
  
pub fn busca_examples() {  
  // ...  
}  
  
pub fn busca(lst: List(Par(a, b), chave: a) -> Result(b, Nil) {  
  case lst {  
    [] -> Error(Nil)  
    [primeiro, ..] if primeiro.chave == chave -> Ok(primeiro.valor)  
    [_, ..resto] -> busca(resto, chave)  
  }  
}
```

Projete uma função que junte todos os elementos de uma lista de strings (não vazias) separando-os com “, ” ou/e “ e ”, de acordo com a gramática do Português.

Exemplos: junta com “,” e “e”

```
/// Parece difícil escrever o propósito... Vamos fazer os exemplos primeiro.  
pub fn junta_virgula_e(lst: List(String)) -> String { todo }
```

Exemplos

```
junta_virgula_e([]) → ""
```

```
junta_virgula_e(["maçã"]) → "maçã"
```

```
junta_virgula_e(["banana", "maçã"]) → "banana e maçã"
```

```
junta_virgula_e(["mamão", "banana", "maçã"]) → "mamão, banana e maçã"
```

```
junta_virgula_e(["aveia", "mamão", "banana", "maçã"]) → "aveia, mamão, banana e maçã"
```

Em todos os exemplos as respostas são calculadas da mesma forma? Não! Os três primeiros exemplos tem uma forma específica, que não é recursiva. Então precisamos criar três casos base.

Exemplos: junta com “,” e “e”

```
/// Produz uma string juntando os elementos de *lst* da seguinte forma:  
/// - Se a *lst* é vazia, devolve "".  
/// - Se a *lst* tem apenas um elemento, devolve esse elemento.  
/// - Senão, junta as strings de *lst*, separando-as com ", ", com exceção  
/// da última string, que é separada com " e ".  
pub fn junta_virgula_e(lst: List(String)) -> String {  
    todo  
}  
  
pub fn junta_virgula_e_examples() {  
    check.eq(junta_virgula_e([]), "")  
    check.eq(junta_virgula_e(["maçã"]), "maçã")  
    check.eq(junta_virgula_e(["mamão", "banana", "maçã"]), "mamão, banana e maçã")  
    check.eq(junta_virgula_e(["aveia", "mamão", "banana", "maçã"]),  
        "aveia, mamão, banana e maçã")  
}
```

```
pub fn junta_virgula_e(lst: List(String)) -> String {  
  case lst {  
    [] -> todo  
    [primeiro] -> todo  
    [primeiro, segundo] -> todo  
    [primeiro, ..resto] -> todo  
  }  
}
```

```
pub fn junta_virgula_e(lst: List(String)) -> String {  
  case lst {  
    [] -> ""  
    [primeiro] -> primeiro  
    [primeiro, segundo] -> primeiro <> " e " <> segundo  
    [primeiro, ..resto] -> primeiro <> ", " <> junta_virgula_e(resto)  
  }  
}
```

Exemplos: junta com “,” e “e” em Python

```
def junta_virgula_e(lst: str) -> str:
    match lst:
        case []:
            return ''
        case [primeiro]:
            return primeiro
        case [primeiro, segundo]:
            return primeiro + ' e ' + segundo
        case _:
            return lst[0] + ', ' + junta_virgula_e(lst[1:])
```


Revisão

Usamos tipos com autorreferências quando queremos representar dados de tamanhos arbitrários.

- Usamos funções recursivas para processar dados de tipos com autorreferências.

Para ser bem formada, uma definição com autorreferência deve ter:

- Pelo menos um caso base (sem autorreferência): são utilizados para criar os valores iniciais
- Pelo menos um caso com autorreferência: são utilizados para criar novos valores a partir de valores existentes

Referências

Básicas

- Vídeos Self-Reference
- Vídeos Naturals
- Capítulos 8 a 12 do livro HTDP
- Seções 2.3, 2.4 e 3.8 do Guia Racket

Complementares

- Seções 2.1 (2.1.1 - 2.1.3) e 2.2 (2.2.1) do livro SICP
- Seções 3.9 da Referência Racket
- Seção 6.3 do livro TSPL4