Autorreferência e recursividade

Parte II

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Números Naturais

Um número natural é atômico ou composto?

- Atômico quando usado em operações aritméticas, comparações, etc;
- Composto quando uma iteração precisa ser feita com base no valor do número.

Se um número natural pode ser visto como um dado composto

- Quais são as partes que compõem o número?
- Como (de)compor um número?

Definição

Um número **natural** é

- `0`; ou
- `n + 1` onde *n* é um número **natural**

Com base nesta definição, criamos um modelo para funções com números naturais.

```
fn fn_para_natural(n: Int) {
  case n {
    0 -> todo
    _ -> {
      todo
      n
      fn_para_natural(n - 1)
    }
  }
}
```

Qual o problema desse modelo? Se `n` não é zero, ele pode ser negativo e a recursão não terminaria. O problema é que o Gleam não possui números naturais.

Definição

Um número **natural** é

- `0`; ou
- `n + 1` onde *n* é um número **natural**

Com base nesta definição, criamos um modelo para funções com números naturais.

```gleam
fn fn_para_natural(n: Int) {
  case n {
    // Necessário porque o Gleam
    // não possui números naturais
    _ if n < 0 -> todo
    0 -> todo
    _ -> {
      todo
      n
      fn_para_natural(n - 1)
    }
  }
}
```

Exemplo: soma naturais

Dado um número natural n, defina uma função que some os números naturais menores ou iguais a n.

Exemplo: soma naturais

```
/// Devolve a soma 1 + 2 + ... + n.
fn soma_nat(n: Int) -> Int {
  todo
}
```

```
fn soma_nat_examples() {
  check.eq(soma_nat(-1), 0)
  check.eq(soma_nat(0), 0)
  check.eq(soma_nat(1), 1)
  check.eq(soma_nat(3), 6)
  check.eq(soma_nat(4), 10)
}
```

Exemplo: soma naturais

```
/// Devolve a soma 1 + 2 + ... + n.
fn soma_nat(n: Int) -> Int {
  case n {
    _ if n < 0 -> todo
    0 -> todo
    _ -> {
      todo
      n
      soma_nat(n - 1)
    }
  }
}
```

```
fn soma_nat_examples() {
  check.eq(soma_nat(-1), 0)
  check.eq(soma_nat(0), 0)
  check.eq(soma_nat(1), 1)
  check.eq(soma_nat(3), 6)
  check.eq(soma_nat(4), 10)
}
```

Exemplo: soma naturais

```
/// Devolve a soma 1 + 2 + ... + n.
fn soma_nat(n: Int) -> Int {
  case n {
    _ if n <= 0 -> 0
    _ -> {
      todo
      n
      soma_nat(n - 1)
    }
  }
}
```

```
fn soma_nat_examples() {
  check.eq(soma_nat(-1), 0)
  check.eq(soma_nat(0), 0)
  check.eq(soma_nat(1), 1)
  check.eq(soma_nat(3), 6)
  check.eq(soma_nat(4), 10)
}
```

Exemplo: soma naturais

```gleam
/// Devolve a soma 1 + 2 + ... + n.
fn soma_nat(n: Int) -> Int {
  case n {
    _ if n <= 0 -> 0
    _ -> n + soma_nat(n - 1)
  }
}
```

```gleam
fn soma_nat_examples() {
  check.eq(soma_nat(-1), 0)
  check.eq(soma_nat(0), 0)
  check.eq(soma_nat(1), 1)
  check.eq(soma_nat(3), 6)
  check.eq(soma_nat(4), 10)
}
```

Exemplo: lista de números

Dado um número natural *n*, defina uma função que devolva `[1, 2, ..., n - 1, n]`.

Exemplo: lista de números

```gleam
/// Cria uma lista com os valores
/// 1, 2, ..., n-1, n.
fn lista_num(n: Int) -> List(Int) {
  todo
}
```

```gleam
fn lista_num_examples() {
  check.eq(lista_num(-1), [])
  check.eq(lista_num(0), [])
  check.eq(lista_num(1), [1])
  check.eq(lista_num(2), [1, 2])
  check.eq(lista_num(3), [1, 2, 3])
}
```

Exemplo: lista de números

```gleam
/// Cria uma lista com os valores
/// 1, 2, ..., n-1, n.
fn lista_num(n: Int) -> List(Int) {
  case n {
    _ if n < 0 -> todo
    0 -> todo
    _ -> {
      todo
      n
      lista_num(n - 1)
    }
  }
}
```

```gleam
fn lista_num_examples() {
  check.eq(lista_num(-1), [])
  check.eq(lista_num(0), [])
  check.eq(lista_num(1), [1])
  check.eq(lista_num(2), [1, 2])
  check.eq(lista_num(3), [1, 2, 3])
}
```

Exemplo: lista de números

```gleam
/// Cria uma lista com os valores
/// 1, 2, ..., n-1, n.
fn lista_num(n: Int) -> List(Int) {
  case n {
    _ if n <= 0 -> []
    _ -> {
      todo
      n
      lista_num(n - 1)
    }
  }
}
```

```gleam
fn lista_num_examples() {
  check.eq(lista_num(-1), [])
  check.eq(lista_num(0), [])
  check.eq(lista_num(1), [1])
  check.eq(lista_num(2), [1, 2])
  check.eq(lista_num(3), [1, 2, 3])
}
```

Exemplo: lista de números

```gleam
/// Cria uma lista com os valores
/// 1, 2, ..., n-1, n.
fn lista_num(n: Int) -> List(Int) {
  case n {
    _ if n <= 0 -> []
    _ -> adiciona_fim(lista_num(n - 1), n)
  }
}
```

```gleam
fn lista_num_examples() {
  check.eq(lista_num(-1), [])
  check.eq(lista_num(0), [])
  check.eq(lista_num(1), [1])
  check.eq(lista_num(2), [1, 2])
  check.eq(lista_num(3), [1, 2, 3])
}
```

Exemplo: adiciona fim

```gleam
/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(
  lst: List(Int),
  n: Int,
) -> List(Int) {
  todo
}
```

```gleam
fn adiciona_fim_examples() {
  check.eq(adiciona_fim([], 3), [3])
  check.eq(adiciona_fim([3], 4), [3, 4])
  check.eq(adiciona_fim([3, 4], 1), [3, 4, 1])
}
```

Exemplo: adiciona fim

```gleam
/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(
  lst: List(Int),
  n: Int,
) -> List(Int) {
  case lst {
    [] -> { todo n }
    [primeiro, ..resto] -> {
      todo
      n
      primeiro
      adiciona_fim(resto, n)
    }
  }
}
```

```gleam
fn adiciona_fim_examples() {
  check.eq(adiciona_fim([], 3), [3])
  check.eq(adiciona_fim([3], 4), [3, 4])
  check.eq(adiciona_fim([3, 4], 1), [3, 4, 1])
}
```

Exemplo: adiciona fim

```gleam
/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(
  lst: List(Int),
  n: Int,
) -> List(Int) {
  case lst {
    [] -> [n]
    [primeiro, ..resto] -> {
      todo
      n
      primeiro
      adiciona_fim(resto, n)
    }
  }
}
```

```gleam
fn adiciona_fim_examples() {
  check.eq(adiciona_fim([], 3), [3])
  check.eq(adiciona_fim([3], 4), [3, 4])
  check.eq(adiciona_fim([3, 4], 1), [3, 4, 1])
}
```

Exemplo: adiciona fim

```gleam
/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(
  lst: List(Int),
  n: Int,
) -> List(Int) {
  case lst {
    [] -> [n]
    [primeiro, ..resto] ->
      [primeiro,
       ..adiciona_fim(resto, n)]
  }
}
```

```gleam
fn adiciona_fim_examples() {
  check.eq(adiciona_fim([], 3), [3])
  check.eq(adiciona_fim([3], 4), [3, 4])
  check.eq(adiciona_fim([3, 4], 1), [3, 4, 1])
}
```

Exemplo: adiciona fim

```gleam
/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(
  lst: List(a),
  n: a,
) -> List(a) {
  case lst {
    [] -> [n]
    [primeiro, ..resto] ->
      [primeiro,
       ..adiciona_fim(resto, n)]
  }
}
```

```gleam
fn adiciona_fim_examples() {
  check.eq(adiciona_fim([], 3), [3])
  check.eq(adiciona_fim([3], 4), [3, 4])
  check.eq(adiciona_fim([3, 4], 1), [3, 4, 1])
}
```

Inteiros

Às vezes, queremos utilizar um caso base diferente de 0.

Podemos generalizar a definição de número natural para incluir um limite inferior diferente de 0.

Definição Inteiro

Um número **inteiro menor ou igual a x** é

- x; ou
- n + 1 onde n é um número **inteiro menor ou igual a x**.

```
fn fn_para_inteiro_lt_x(n: Int) {
  case n {
    _ if n < x -> todo
    _ if n == x -> todo
    _ -> {
      todo
      x
      fn_para_inteiro_lt_x(n - 1)
    }
  }
}
```

Árvores binárias

Como podemos definir uma árvore binária?

Árvores binárias

Uma **árvore binária** é

- Vazia; ou
- Um nó contendo um valor e **árvores binárias** à esquerda e à direita.

```
type Arvore(a) {
  Vazia
  No(valor: a, esq: Arvore(a), dir: Arvore(a))
}
```


```
No(3,
  No(4,
    No(3, Vazia, Vazia)
    Vazia),
  No(7,
    No(8, Vazia, Vazia)
    No(9,
      No(10, Vazia, Vazia)
      Vazia)))
```

Árvores binárias

Uma **árvore binária** é

- Vazia; ou
- Um nó contendo um valor e **árvores binárias** à esquerda e à direita.

```
type Arvore(a) {
  Vazia
  No(valor: a, esq: Arvore(a), dir: Arvore(a))
}
```

Modelo de função para árvores binárias

```
fn fn_para_ab(r: Arvore(a)) {
  case arv {
    Vazia -> todo
    No(valor, esq, dir) -> {
      todo
      valor
      fn_para_ab(esq)
      fn_para_ab(dir)
    }
  }
}
```

Exemplo: nós folhas

Projete uma função que determine a quantidade de nós-folha em uma árvore.

Exemplo: nós folhas

```
/// Determina o número de nós-folha de *r*.
fn num_folhas(r: Arvore(a)) -> Int {
  todo
}
```

```
fn num_folhas_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     / \   / \
  //    3   2 8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), No(2, Vazia, Vaz
  let t4 = No(3, t3, t2)
  check.eq(num_folhas(Vazia), 0)
  check.eq(num_folhas(t0), 1)
  check.eq(num_folhas(t1), 1)
  check.eq(num_folhas(t2), 2)
  check.eq(num_folhas(t3), 2)
  check.eq(num_folhas(t4), 4)
}
```

Exemplo: nós folhas

```
/// Determina o número de nós-folha de *r*.
fn num_folhas(r: Arvore(a)) -> Int {
  case r {
    Vazia -> todo
    No(valor, esq, dir) -> {
      todo
      valor
      num_folhas(esq)
      num_folhas(dir)
    }
  }
}
```

```
fn num_folhas_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     / \   / \
  //    3   2 8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), No(2, Vazia, Vaz
  let t4 = No(3, t3, t2)
  check.eq(num_folhas(Vazia), 0)
  check.eq(num_folhas(t0), 1)
  check.eq(num_folhas(t1), 1)
  check.eq(num_folhas(t2), 2)
  check.eq(num_folhas(t3), 2)
  check.eq(num_folhas(t4), 4)
}
```

Exemplo: nós folhas

```
/// Determina o número de nós-folha de *r*.
fn num_folhas(r: Arvore(a)) -> Int {
  case r {
    Vazia -> 0
    No(valor, esq, dir) -> {
      todo
      valor
      num_folhas(esq)
      num_folhas(dir)
    }
  }
}
```

```
fn num_folhas_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     / \   / \
  //    3   2 8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), No(2, Vazia, Vaz
  let t4 = No(3, t3, t2)
  check.eq(num_folhas(Vazia), 0)
  check.eq(num_folhas(t0), 1)
  check.eq(num_folhas(t1), 1)
  check.eq(num_folhas(t2), 2)
  check.eq(num_folhas(t3), 2)
  check.eq(num_folhas(t4), 4)
}
```

Exemplo: nós folhas

```
/// Determina o número de nós-folha de *r*.
fn num_folhas(r: Arvore(a)) -> Int {
  case r {
    Vazia -> 0
    No(_, esq, dir) ->
      case esq, dir {
        Vazia, Vazia -> 1
        _, _ ->
          num_folhas(esq) + num_folhas(dir)
      }
  }
}
```

```
fn num_folhas_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     / \   / \
  //    3   2 8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), No(2, Vazia, Vaz
  let t4 = No(3, t3, t2)
  check.eq(num_folhas(Vazia), 0)
  check.eq(num_folhas(t0), 1)
  check.eq(num_folhas(t1), 1)
  check.eq(num_folhas(t2), 2)
  check.eq(num_folhas(t3), 2)
  check.eq(num_folhas(t4), 4)
}
```

Exemplo: nós folhas

```
/// Determina o número de nós-folha de *r*.
fn num_folhas(r: Arvore(a)) -> Int {
  case r {
    Vazia -> 0
    No(_, Vazia, Vazia) -> 1
    No(_, esq, dir) ->
      num_folhas(esq) + num_folhas(dir)
  }
}
```

```
fn num_folhas_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     / \   / \
  //    3   2 8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), No(2, Vazia, Vaz
  let t4 = No(3, t3, t2)
  check.eq(num_folhas(Vazia), 0)
  check.eq(num_folhas(t0), 1)
  check.eq(num_folhas(t1), 1)
  check.eq(num_folhas(t2), 2)
  check.eq(num_folhas(t3), 2)
  check.eq(num_folhas(t4), 4)
}
```

Exemplo: altura árvore

Defina uma função que determine a altura de uma árvore binária. A altura de uma árvore binária é a distância entre a raiz e o seu descendente mais afastado. Uma árvore com um único nó tem altura 0.

Exemplo: altura árvore

```gleam
/// Devolve a altura de *r*. A altura de uma
/// árvore binária é a distância da raiz a seu
/// descendente mais afastado. Uma árvore com
/// um único nó tem altura 0.
fn altura(r: Arvore(a)) -> Int {
  todo
}
```

```gleam
fn altura_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     /     / \
  //    3     8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), Vazia)
  let t4 = No(3, t3, t2)
  check.eq(altura(Vazia), todo)
  check.eq(altura(t0), 0)
  check.eq(altura(t1), 1)
  check.eq(altura(t2), 2)
  check.eq(altura(t3), 1)
  check.eq(altura(t4), 3)
}
```

Exemplo: altura árvore

```gleam
/// Devolve a altura de *r*. A altura de uma
/// árvore binária é a distância da raiz a seu
/// descendente mais afastado. Uma árvore com
/// um único nó tem altura 0.
fn altura(r: Arvore(a)) -> Int {
  case r {
    Vazia -> todo
    No(valor, esq, dir) -> {
      todo
      valor
      altura(esq)
      altura(dir)
    }
  }
}
```

```gleam
fn altura_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     /     / \
  //    3     8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), Vazia)
  let t4 = No(3, t3, t2)
  check.eq(altura(Vazia), todo)
  check.eq(altura(t0), 0)
  check.eq(altura(t1), 1)
  check.eq(altura(t2), 2)
  check.eq(altura(t3), 1)
  check.eq(altura(t4), 3)
}
```

Exemplo: altura árvore

```
/// Devolve a altura de *r*. A altura de uma
/// árvore binária é a distância da raiz a seu
/// descendente mais afastado. Uma árvore com
/// um único nó tem altura 0.
fn altura(r: Arvore(a)) -> Int {
  case r {
    Vazia -> todo
    No(_, esq, dir) ->
      1 + int.max(altura(esq), altura(dir))
    }
  }
}
```

```
fn altura_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     /     / \
  //    3     8   9  t1
  //              /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), Vazia)
  let t4 = No(3, t3, t2)
  check.eq(altura(Vazia), todo)
  check.eq(altura(t0), 0)
  check.eq(altura(t1), 1)
  check.eq(altura(t2), 2)
  check.eq(altura(t3), 1)
  check.eq(altura(t4), 3)
}
```

Exemplo: altura árvore

```gleam
/// Devolve a altura de *r*. A altura de uma
/// árvore binária é a distância da raiz a seu
/// descendente mais afastado. Uma árvore com
/// um único nó tem altura 0 e uma árvore vazia
/// tem altura -1.
fn altura(r: Arvore(a)) -> Int {
  case r {
    Vazia -> -1
    No(_, esq, dir) ->
      1 + int.max(altura(esq), altura(dir))
  }
}
```

```gleam
fn altura_examples() {
  //     t4  3
  //       /   \
  //  t3  4     7  t2
  //     /     / \
  //    3     8   9  t1
  //             /
  //         t0  10
  let t0 = No(10, Vazia, Vazia)
  let t1 = No(9, t0, Vazia)
  let t2 = No(7, No(8, Vazia, Vazia), t1)
  let t3 = No(4, No(3, Vazia, Vazia), Vazia)
  let t4 = No(3, t3, t2)
  check.eq(altura(Vazia), -1)
  check.eq(altura(t0), 0)
  check.eq(altura(t1), 1)
  check.eq(altura(t2), 2)
  check.eq(altura(t3), 1)
  check.eq(altura(t4), 3)
}
```

Árvores

Projete um tipo de dado para representar um diretório ou arquivo em um sistema de arquivos.

```
disciplinas/
+- 12026/
|  +- alunos.txt
|  +- trabs/
|     +- trab1.md
|     +- correcoes/
|     |  +- rascunho.txt
|     |  +- final.txt
|     +- trab2.md
+- 6879/
+- 6884/
+- anotacoes.txt
```

Árvores

Uma **entrada** no sistema de arquivos é:

- Um arquivo com um nome; ou
- Um diretório com um nome e uma **lista de entradas**.

Uma **lista de entradas** é:

- Vazia; ou
- Um par com o primeiro e o resto, onde o primeiro é uma **entrada** e o resto é uma **lista de entradas**.

```
type Entrada {
  Arq(String)
  Dir(String, List(Entrada))
}
```

Árvores

```
disciplinas/
+- 12026/
|  +- alunos.txt
|  +- trabs/
|     +- trab1.md
|     +- correcoes/
|     |  +- rascunho.txt
|     |  +- final.txt
|     +- trab2.md
+- 6879/
+- 6884/
+- anotacoes.txt
```

```
Dir("disciplinas", [
  Dir("12026", [
    Arq("alunos.txt"),
    Dir("trabs", [
      Arq("trab1.md"),
      Dir("correcoes", [
        Arq("rascunho.txt"),
        Arq("final.txt")
      ]),
      Arq("trab2.md"),
    ]),
  ]),
  Dir("6879", []),
  Dir("6884", []),
  Arq("anotacoes.txt"),
])
```

Árvores

Uma **entrada** no sistema de arquivos é:

- Um arquivo com um nome; ou
- Um diretório com um nome e uma **lista de entradas**.

Uma **lista de entradas** é:

- Vazia; ou
- Um par com o primeiro e o resto, onde o primeiro é uma **entrada** e o resto é uma **lista de entradas**.

```
fn fn_para_entrada(ent: Entrada) {
  case ent {
    Arq(nome) -> { todo nome }
    Dir(nome, entradas) -> {
      todo nome
           fn_para_entradas(entradas)
    }
  }
}

fn fn_para_entradas(entradas: List(Entrada)) {
  case entradas {
    [] -> todo
    [primeiro, ..resto] -> {
      todo fn_para_entrada(primeiro)
           fn_para_entradas(resto)
    }
  }
}
```

Exemplo: arquivos txt

Projete uma função para encontrar os caminhos para todos os arquivos `.txt`.

Exemplo: arquivos txt

```
disciplinas/
+- 12026/
|  +- alunos.txt
|  +- trabs/
|     +- trab1.md
|     +- correcoes/
|     |  +- rascunho.txt
|     |  +- final.txt
|     +- trab2.md
+- 6879/
+- 6884/
+- anotacoes.txt

disciplinas/12026/alunos.txt
disciplinas/12026/trabs/correcoes/rascunho.txt
disciplinas/12026/trabs/correcoes/final.txt
disciplinas/anotacoes.txt
```

```gleam
fn encontra_txt(ent: Entrada) -> List(String) {
  todo
}
```

Exemplo: arquivos txt

```
disciplinas/
+- 12026/
|  +- alunos.txt
|  +- trabs/
|     +- trab1.md
|     +- correcoes/
|     |  +- rascunho.txt
|     |  +- final.txt
|     +- trab2.md
+- 6879/
+- 6884/
+- anotacoes.txt

disciplinas/12026/alunos.txt
disciplinas/12026/trabs/correcoes/rascunho.txt
disciplinas/12026/trabs/correcoes/final.txt
disciplinas/anotacoes.txt
```

```
fn encontra_txt(ent: Entrada) -> List(String) {
  case ent {
    Arq(nome) -> { todo nome }
    Dir(nome, entradas) -> {
      todo nome
           encontra_txt_lista(entradas)
    }
  }
}
fn encontra_txt_lista(entradas: List(Entrada)) -> List(
  case entradas {
    [] -> todo
    [ent, ..resto] -> {
      todo encontra_txt(ent)
           encontra_txt_lista(resto)
    }
  }
}
```

Exemplo: arquivos txt

```
disciplinas/
+- 12026/
|  +- alunos.txt
|  +- trabs/
|     +- trab1.md
|     +- correcoes/
|     |  +- rascunho.txt
|     |  +- final.txt
|     +- trab2.md
+- 6879/
+- 6884/
+- anotacoes.txt

disciplinas/12026/alunos.txt
disciplinas/12026/trabs/correcoes/rascunho.txt
disciplinas/12026/trabs/correcoes/final.txt
disciplinas/anotacoes.txt
```

```
fn encontra_txt(ent: Entrada) -> List(String) {
  case ent {
    Arq(nome) -> case string.ends_with(nome, ".txt") {
      False -> []
      True -> [nome] }
    Dir(nome, entradas) -> {
      todo nome
           encontra_txt_lista(entradas)
    }
  }
}
fn encontra_txt_lista(entradas: List(Entrada)) -> List(
  case entradas {
    [] -> []
    [ent, ..resto] -> {
      todo encontra_txt(ent)
           encontra_txt_lista(resto)
    }
  }
}
```

Exemplo: arquivos txt

```
disciplinas/
+- 12026/
|  +- alunos.txt
|  +- trabs/
|     +- trab1.md
|     +- correcoes/
|     |  +- rascunho.txt
|     |  +- final.txt
|     +- trab2.md
+- 6879/
+- 6884/
+- anotacoes.txt

disciplinas/12026/alunos.txt
disciplinas/12026/trabs/correcoes/rascunho.txt
disciplinas/12026/trabs/correcoes/final.txt
disciplinas/anotacoes.txt
```

```
fn encontra_txt(ent: Entrada) -> List(String) {
  case ent {
    Arq(nome) ->
      case string.ends_with(nome, ".txt") {
        False -> []
        True -> [nome]
      }
    Dir(nome, entradas) ->
      adiciona_prefixo(nome,
                       encontra_txt_lista(entradas))
  }
}
fn encontra_txt_lista(entradas: List(Entrada)) -> List(
  case entradas {
    [] -> []
    [ent, ..resto] ->
      list.append(encontra_txt(ent),
                  encontra_txt_lista(resto))
  }
}
```

Limitações

Limitações

Cada tipo com autorreferência tem um modelo de função que podemos usar como ponto de partida para implementar funções que processam esse tipo de dado.

Embora o modelo seja um ponto de partida, em algumas situações ele pode não ser útil.

Considere o problema de verificar se uma lista de números é um palíndromo (a lista tem os mesmos elementos quando lida da direita para a esquerda e da esquerda para a direita).

Para verificar se `[5, 4, 1, 4]` é um palíndromo, o modelo sugere verificar se `[4, 1, 4]` é um palíndromo.

Como a verificação se `[4, 1, 4]` é um palíndromo pode nos ajudar a determinar se `[5, 4, 1, 4]` é um palíndromo? Ou seja, a solução para o resto pode nos ajudar a compor o resultado para o todo? Não pode...

Considere o problema de verificar se um número natural n é primo (tem exatamente dois divisores distintos, 1 e n).

Para verificar se $n = 13$ é primo, o modelo sugere verificar se 12 é primo.

Como a verificação se 12 é primo pode nos ajudar a determinar se 13 é primo? Não pode...

O problema nos dois casos é o mesmo: a solução do problema original não pode ser obtida a partir da solução do subproblema gerado pela **decomposição estrutural** do dado.

Como proceder nesse caso? Temos algumas opções:

- Redefinimos o problema de forma que a solução para o subproblema estrutural possa ser usada na construção da solução do problema original;
- Fazemos uma decomposição em subproblema(s) de maneira não estrutural e utilizamos a solução desse(s) subproblema(s) para construir a solução do problema original;
- Criamos um plano (sequência de etapas) para construir a solução sem necessariamente pensar na decomposição da entrada em subproblemas do mesmo tipo.

Para o problema do número primo, podemos reescrever o problema da seguinte forma: Dados dois números naturais n e $a \leq n$, projete uma função que determine a quantidade de divisores de n que são $\leq a$.

Se temos a quantidade de divisores de n que são $\leq a - 1$, como obtemos a quantidade de divisores de n que são $\leq a$? Somando 1 se a é divisor de n.

Como podemos utilizar essa função para determinar se um número n é primo? Com a expressão `num_divisores(n, n) == 2`

Número primo

```
/// Produz True se *n* é um número primo,
/// isto é, tem exatamente dois divisores
/// positivos distintos (1 e *n*).
/// Produz False caso contrário.
fn primo(n: Int) -> Bool {
  num_divisors(n, n) == 2
}

/// Calcula o número de divisores positivos
/// de *n* que são menores ou iguais à *a*.
fn num_divisors(n: Int, a: Int) -> Int {
  case a {
    _ if a <= 0 -> 0
    _ if n % a == 0 -> 1 + num_divisors(n, a - 1)
    _ -> num_divisors(n, a - 1)
  }
}
```

```
fn primo_examples() {
  check.eq(primo(1), False)
  check.eq(primo(2), True)
  check.eq(primo(3), True)
  check.eq(primo(4), False)
  check.eq(primo(5), True)
  check.eq(primo(6), False)
  check.eq(primo(7), True)
  check.eq(primo(8), False)
}
```

Para o problema da lista palíndromo, vamos considerar a entrada `[4, 1, 5, 1, 4]`.

Como podemos obter um subproblema da entrada de maneira que a resposta para o subproblema possa nos ajudar a compor a resposta para o problema original? Removendo o primeiro e último elemento da lista.

Se sabemos que uma lista `lst` sem o primeiro e o último elemento é um palíndromo, como determinar se `lst` é um palíndromo? Verificando se o primeiro e o último elemento de `lst` são iguais.

Palíndromo 1

```gleam
/// Produz True se *lst* é um palíndromo, isto é, tem os mesmos elementos quando lida
/// da direita para a esquerda e da esquerda para a direita. Produz False caso contrário.
fn palindromo(lst: List(Int)) -> Bool {
  case lst {
    [] | [_] -> True
    [primeiro, ..] ->
      Ok(primeiro) == list.last(lst) && palindromo(sem_extremos(lst))
  }
}

fn palindromo_examples() {
  check.eq(palindromo([]), True)
  check.eq(palindromo([2]), True)
  check.eq(palindromo([1, 2]), False)
  check.eq(palindromo([3, 3]), True)
  check.eq(palindromo([3, 7, 3]), True)
  check.eq(palindromo([3, 7, 3, 3]), False)
}
```

Exercício: implemente a função `sem_extremos`.

Funções recursivas que operam em subproblemas obtidos pela decomposição estrutural dos dados são chamadas de **funções recursivas estruturais**.

Funções recursivas que operam em subproblemas arbitrários (não estruturais) são chamadas de **funções recursivas generativas**.

O projeto de funções recursivas generativas pode requerer um "*insight*" e por isso tentamos primeiramente resolver os problemas com recursão estrutural.

Plano

Ainda para o problema da lista palíndromo, em vez de pensarmos em decompor o problema em um subproblema da mesma natureza, podemos pensar em um plano, uma sequência de etapas que resolva problemas intermediários mas que gerem o resultado que estamos esperando no final.

Por exemplo, podemos, primeiramente, inverter a lista e depois verificar se a lista de entrada e a lista invertida são iguais.

Note que para este caso precisaríamos projetar duas novas funções. Essas funções poderiam ser implementadas usando recursão estrutural.

Palíndromo 2

```gleam
/// Produz True se *lst* é um palíndromo, isto é, tem os mesmos elementos quando lida
/// da direita para a esquerda e da esquerda para a direita. Produz False caso contrário.
fn palindromo(lst: List(Int)) -> Bool {
  lst == list.reverse(lst)
}

fn palindromo2_examples() {
  check.eq(palindromo([]), True)
  check.eq(palindromo([2]), True)
  check.eq(palindromo([1, 2]), False)
  check.eq(palindromo([3, 3]), True)
  check.eq(palindromo([3, 7, 3]), True)
  check.eq(palindromo([3, 7, 3, 3]), False)
}
```

Exercício: implemente a função `reverse`.

Revisão

Usamos tipos com autorreferência 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 usados para criar os valores iniciais;
- Pelo menos um caso com autorreferência: são usados para criar novos valores a partir de valores existentes.

Às vezes é interessante pensar em números inteiros e naturais como sendo compostos e definidos com autorreferência.

Existem dois tipos de recursão: estrutural e generativa.

- A recursão estrutural é aquela feita na decomposição natural do dado (para as partes que são autorreferências na definição do dado).
- A recursão generativa é aquela que não é estrutural.

A recursão estrutural só pode ser utilizada quando a solução do problema pode ser expressa em termos da solução do subproblema estrutural. Para os demais problemas podemos tentar três abordagens:

- Alterar o problema e utilizar recursão estrutural;
- Usar recursão generativa;
- Usar um plano (sequência de etapas).

Referências

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