Universidade Estadual de Maringá - Departamento de Informática

Programação Funcional

RESUMO DA SINTAXE DE GLEAM - V3

Sintaxe geral

```
// importação
import gleam/módulo

// definição de função
pub fn nome(nome1: Tipo, …) -> Tipo {
  expressão
  expressão
  …
}

// chamada de função
nome(expressão1, expressão2, …)
```

```
// seleção
…
  case expressão1, expressão2, … {
    padrão1, padrão2, … if condição -> expressão
    padrão1, padrão2, … -> expressão
    padrão1, _, … -> expressão
    …
    _, _, … -> expressão
  }
…

// definição de tipo
pub type Nome() {
  VarianteA(campoA1: Tipo, …)
  VarianteB(campoB1: Tipo, …)
  …
}
```

Exemplo de criação de estrutura e casamento de padrões

```
pub type P{
  P(x: Int, y: Int)
}

pub fn main() {
  let p1 = P(1, 2)        // criação normal
  let p2 = P(..p1, y:3)   // criação com cópia parcial (atualização): p2 é P(1, 3)

  let P(a, b) = p1        // casamento de padrão (desestruturação): a é 1, b é 2
  let P(c, _) = p1        // c é 1
  let P(d, ..) = p1       // d é 1
  let P(y:, ..) = p1      // y é 2
  let P(y:e, ..) = p1     // e é 2

  case p1 {
      P(a, b) -> a + b    // casamento de padrão (seleção): a é 1, b é 2
      P(c, _) -> c        // c é 1
      P(d, ..) -> d       // d é 1
      P(y:, ..) -> y      // y é 2
      P(y:e, ..) -> e     // e é 2
    }
}
```

Testes

```
import sgleam/check

pub fn nome_examples() {
  check.eq(nome(expressão1, …), valor1)
  check.eq(nome(expressão2, …), valor2)
  …
}
```

Exemplo de tipo opcional

```
import gleam/option.{type Option, Some, None}

pub fn soma_um(a: Option(Int)) -> Option(Int) {
  case a {
    None -> None
    Some(x) -> Some(x + 1)
  }
}
```

Exemplo de tipo resultado

```
> int.parse("123")
Ok(123)

> int.parse("45.6")
Error(Nil)
```

```
import gleam/int

pub fn soma_strings(a: String, b: String) -> Result(String, Nil) {
  case int.parse(a), int.parse(b) {
    Ok(x), Ok(y) -> Ok(int.to_string(x + y))
    _, _ -> Error(Nil)
  }
}
```

Operações com listas

```
> let lst0 = [] // lista vazia
[]

> let lst1: List(Int) = [3, 8]
[3, 8]

// nova lista a partir de outra
> let lst2 = [7, ..lst1]
[7, 3, 8]
```

```
case lst {
  [] -> …        // se a lista é vazia
  [a] -> …       // se a lista tem um elemento
  [b, c] -> …    // se a lista tem dois elementos
  [p, ..r] -> … // caso geral:
                 // p é o primeiro elemento
                 // r é o resto, sempre é uma lista
  [x, ..] -> …  // ignora o resto
  [_, ..y] -> … // ignora o primeiro
}
```

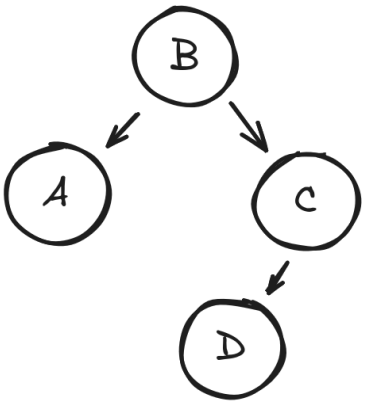
Modelo de função para lista

```
pub fn nome(lst: Lista(Tipo), nome1: Tipo, …) -> Tipo {
  case lst {
    [] -> { … }               // caso base (podem haver outros)
    [primeiro, ..resto] -> { // caso geral que contém a recursão (podem haver outros)
      …
      nome(resto, …)
    }
  }
}
```

Definição de árvore binária

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

pub const arvore = No(
  "B",
  No("A", Vazia, Vazia),
  No("C",
     No("D", Vazia, Vazia),
     Vazia))
```



Modelo de função para árvore binária

```
pub fn nome(arv: Arvore(Tipo), nome1: Tipo, …) -> Tipo {
  case arv {
    Vazia -> { … }           // caso base (podem haver outros)
    No(valor, esq, dir) -> { // caso geral que contêm as recursões (podem haver outros)
      …
      nome(esq)
      nome(dir)
    }
  }
}
```