Fundamentos

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

O paradigma de programação funcional é baseado na definição e aplicação de funções.

Uma **função** é uma conjunto de expressões que mapeia valores de entrada para valores de saída.

Uma **expressão** é uma entidade sintática que quando avaliada (reduzida) produz um valor.

Vamos ver uma sequência de definições de expressões e regras de avaliação.

Definição de expressão (versão 0.1)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva

Um **literal** é um valor que é representado diretamente no código. Em geral, os literais são utilizados para criar valores de tipos primitivos.

Uma **tipo primitivo** é um tipo suportado diretamente pela linguagem de programação.

Uma **função primitiva** é uma função suportada diretamente pela linguagem de programação.

Literais de tipos primitivos

Gleam provê 9 tipos primitivos. Todos os nomes de tipos começam com letra maiúscula.

Número inteiro (`Int`)

- `1345`
- `9_876`

Booleano (`Bool`)

- `True`
- `False`

Números de ponto flutuante (`Float`)

- `2.65`
- `2.0e12`
- `7.4e-10`

Strings (`String`)

- `"din uem"`
- `"apenas \"um\" teste"`

Veremos alguns outros tipos primitivos ao longo da disciplina.

Funções primitivas

Gleam provê diversas operações primitivas, a maioria delas está disponível na forma de operadores. Todos os nomes de funções começam com letra minúscula.

Operações com inteiros:

- `+` (`int.add`)
- `-` (`int.subtract`)
- `* / % > >= < <= == !=`
- `int.to_float` e diversas outras no módulo `int`.

Operações com floats:

- `*.` (`float.product`)
- `/.` (`float.divide`)
- `+. -.  >. >=. <. <=. == !=`
- `float.truncate` e diversas outras no módulo `float`.

Operações com strings:

- `<>` (`string.append`)
- `== !=`
- `string.slice` e diversas outras no módulo `string`.

Operações com booleanos:

- `!` (`bool.negate`)
- `== !=`
- Outros operadores que veremos depois.

Processo de avaliação de expressões (versão 0.1)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva

Regra para **avaliação de expressão**

- Literal $\rightarrow$ valor que o literal representa
- Função primitiva $\rightarrow$ sequência de instruções de máquina associada com a função

Como a regra de avaliação de expressão está ligada com a definição de expressão?

Uma expressão é definida em termos de dois casos e por isso a regra de avaliação de expressão também é definida por dois casos.

Exemplo de avaliação de expressões

```
> True
True
> 231
231
> "Banana"
"Banana"
> int.add
//fn(a, b) { ... }
```

A definição de expressão que acabamos de ver parece bastante limitada, o que está faltando?

Uma forma de combinar expressões para formar novas expressões!

Combinações

Alguns exemplos de combinações

```
> { 2 + 12 } * 5
70
> "Gol" <> string.repeat("!", 4)
"Gol!!!!"
> 10 / 0  // Divisão por zero é zero!
0
```

```
> int.multiply(int.add(2, 12), 5)
70
> string.append("Gol",
                string.repeat("!", 4))
"Gol!!!"
> int.divide(10, 0)
0
```

Considerando apenas funções e literais (vamos deixar os operadores de lado), qual é forma de combinar expressões para criar novas expressões?

A chamada de função. Como podemos definir como são formadas as chamadas de funções?

Primeira tentativa

Uma chamada de função começa com uma função primitiva, seguido de abre parêntese, seguido de uma sequência de **literais** separados por vírgula, seguido de fecha parêntese.

Essa definição é adequada? Não!

O exemplo `int.multiply(int.add(2, 12), 5)` não está de acordo com essa definição!

Segunda tentativa

Uma chamada de função começa com uma função primitiva, seguido de abre parêntese, seguido de uma sequência de **expressões** separadas por vírgula, seguido de fecha parêntese.

Vamos usar uma definição mais genérica.

Uma **chamada de função** consiste de uma **expressão** seguido por uma sequência de **expressões** entre parênteses separadas por vírgula.

- A primeira expressão é a **operador**;
- As demais expressões são os **operandos**.

Qual é o valor produzido pela avaliação de uma chamada de função?

- O resultado da aplicação do valor do operador aos valores dos operandos.

Vamos atualizar a definição de expressão para incluir as chamadas de funções.

Definição de expressão (versão 0.2)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva; ou
- Uma chamada de função (**expressão** seguida de uma lista de **expressões** entre parênteses)

Regra para **avaliação de expressão**

- Literal $\longrightarrow$ valor que o literal representa
- Função primitiva $\longrightarrow$ sequência de instruções de máquina associada com a função
- Chamada de função
  - **Avalie cada expressão** da chamada da função, isto é, reduza cada expressão para um valor
    $\longrightarrow$ resultado da aplicação da função aos argumentos

Algumas observações interessantes

- Uma expressão é definida por três casos e a regra de avaliação também tem três casos.
- Quando uma expressão é uma chamada de função, ela contém outras expressões. Quando uma definição refere-se a si mesmo, dizemos que ela é uma definição com **autorreferência**. O uso de autorreferência permite a criação de expressões de tamanhos arbitrários.
- O processo de avaliação para uma expressão que é uma chamada de função requer a chamada do processo de avaliação para suas expressões. Quando um processo é definido em termos de si mesmo, dizemos que ele é **recursivo**. O uso de recursividade permite a avaliação de expressões de tamanhos arbitrários.
- Uma autorreferência em uma definição implicada (geralmente) em uma recursão para processar os elementos que seguem a definição.

Estamos usando os conceitos de autorreferência e recursividade para entender o funcionamento da linguagem Gleam (a estrutura das linguagens de programação são recursivas), mas iremos ver que estes conceitos são fundamentais também para criar programas no paradigma funcional.

Exemplo de avaliação de um expressão

```
import gleam/int.{add, multiply as mul, subtract as sub}

3 * { 2 * 4 + { 3 + 5 } } + { { 10 - 7 } + 6 }

add(mul(3, add(mul(2, 4), add(3, 5))), add(sub(10, 7), 6)) // mul(2, 4) -> 8
add(mul(3, add(8, add(3, 5))), add(sub(10, 7), 6)) // add(3, 5) -> 8
add(mul(3, add(8, 8)), add(sub(10, 7), 6)) // add(8, 8) -> 16
add(mul(3, 16), add(sub(10, 7), 6)) // mul(3, 16) -> 48
add(48, add(sub(10, 7), 6)) // sub(10, 7) -> 3
add(48, add(3, 6)) // add(3, 6) -> 9
add(48, 9) // add(48, 9) -> 57
57
```

Vimos anteriormente que o paradigma de programação funcional é baseado na definição e aplicações de funções.

Como funções são definidas em termos de expressões, nós vimos primeiramente o que são expressões.

Agora vamos ver o que são definições e como fazer definições de novas funções.

Definições

Qual é o propósito das definições?

Definições servem para dar nome a objetos computacionais, sejam dados ou funções.

- É a forma de abstração mais elementar

Definições de constantes

A forma geral para definições de constantes em Gleam é:

```
[pub] const nome [: Tipo] = literal
```

Exemplos

```
const x: Int = 10
pub const y = 20
```

```
> x
10
> y
20
```

Note que a especificação do tipo da constante é opcional. Se o tipo não for especificado, ele é inferido pelo compilador.

Definições de funções

A forma geral para definições de novas funções (**funções compostas**) em Gleam é:

```
[pub] fn nome(parametro1 [: Tipo], parametro2 [: Tipo], ...) [-> Tipo] {
    expressão...
}
```

Exemplos

```
fn quadrado(x: Int) -> Int {
    x * x
}
pub fn soma_quadrados(a: Int, b) {
    quadrado(a) + quadrado(b)
}
```

```
> soma_quadrados(3, 4)
25
```

Note que a especificação dos tipo das entradas e saída são opcionais. Se os tipos não forem especificados, eles são inferidos pelo compilador.

Os nomes usados nas definições são associados com os objetos que eles representam e armazenados em um memória chamada de **ambiente**.

Um programa em Gleam é composto por uma sequência de definições e instruções `import`.

Agora precisamos estender a definição de expressões para incluir nomes e alterar a regra de avaliação de expressões para considerar a chamada de funções compostas.

Modelo de substituição

Definição de expressão (versão 0.3)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva; ou
- Um nome; ou
- Uma chamada de função (**expressão** seguida de uma lista de **expressões** entre parênteses)

Regra para **avaliação de expressão**

- Literal $\rightarrow$ valor que o literal representa
- Função primitiva $\rightarrow$ sequência de instruções de máquina associada com a função
- Nome $\rightarrow$ valor associado com o nome no ambiente
- Chamada de função
  - **Avalie cada expressão** da chamada da função
  - Se o operador é uma função primitiva, aplique a função aos argumentos
  - Senão (o operador é uma função composta) , **avalie** o corpo da função **substituindo** cada ocorrência do parâmetro formal pelo argumento correspondente

Essa forma de calcular o resultado das chamadas das funções compostas é chamada de **modelo de substituição**.

Modelo de substituição

```
fn quadrado(x) { x * x }
fn soma_quadrado(a, b) { quadrado(a) + quadrado(b) }
fn f(a) { soma_quadrados(a + 1, a * 2) }

f(5)                            // Substitui f(5) pelo corpo de f com as ocorrências
                                // do parâmetro a substituídas pelo argumento 5

soma_quadrados(5 + 1, 5 * 2)// Reduz 5 + 1 para o valor 6

soma_quadrados(6, 5 * 2)// Reduz 5 * 2 para o valor 10

soma_quadrados(6, 10)// Substitui soma_quadrados(6, 10) pelo corpo ...

quadrado(6) + quadrado(10)// Substitui quadrado(6) pelo corpo ...

{ 6 * 6 } + quadrado(10)// Reduz 6 * 6 para 36

36 + quadrado(10)// Substitui quadrado(10) pelo corpo ...

36 + { 10 * 10 }// Reduz 10 * 10 para 100

36 + 100// Reduz 36 + 100 para 136

136
```

Ao invés de avaliar os operandos e depois fazer a substituição, existe um outro modo de avaliação que primeiro faz a substituição e apenas avalia os operandos quando (e se) eles forem necessários.

```
f(5)
soma_quadrados(5 + 1, 5 * 2)
quadrado(5 + 1) + quadrado(5 * 2)
{ 5 + 1 } * { 5 + 1} + quadrado(5 * 2)
{ 5 + 1 } * { 5 + 1} + { 5 * 2 } * { 5 * 2 }
6 * { 5 + 1} + { 5 * 2 } * { 5 * 2 }
6 * 6 + { 5 * 2 } * { 5 * 2 }
36 + { 5 * 2 } * { 5 * 2 }
36 + 10 * { 5 * 2 }
36 + 10 * 10
36 + 100
136
```

A resposta gerada por essas duas formas de substituição será sempre a mesma?

Para funções que terminam sim!

Veremos depois os casos para funções que não terminam.

Este método de avaliação alternativo de primeiro substituir e depois reduzir, é chamado de **avaliação em ordem normal** (avaliação preguiçosa).

O método de avaliação que primeiro avalia os argumentos e depois aplica a função é chamado de **avaliação em ordem aplicativa** (avaliação ansiosa).

O Gleam usa por padrão a avaliação em ordem aplicativa.

O Haskell usa a avaliação em ordem normal.

Exercício custo

1. O seu amigo Alan está planejando uma viagem pro final do ano com a família e está considerando diversos destinos. Uma das coisas que ele está levando em consideração é o custo da viagem, que inclui, entre outras coisas, hospedagem, combustível e o pedágio. Para o cálculo do combustível ele pediu a sua ajuda, ele disse que sabe a distância que vai percorrer, o preço do litro do combustível e o rendimento do carro (quantos quilômetros o carro anda com um litro de combustível), mas que é muito chato ficar fazendo o cálculo manualmente, então ele quer que você faça um programa para calcular o custo do combustível em uma viagem.

O que de fato precisa ser feito?

Calcular o custo do combustível (saída) em uma viagem sabendo a distância do percurso, o preço do litro do combustível e o rendimento do carro (entradas).

Como determinar o processo (forma) que a saída é computada a partir da entrada?

Fazendo exemplos específicos e generalizando o processo.

Exemplo de entrada

- Distância: 400.0 Km
- Preço do litro: R$ 5.0
- Rendimento: 10.0 Km/l

Saída

- Quantidade de litros (Distância / Rendimento): `400.0 /. 10.0` $\longrightarrow$ `40.0`
- Custo (Quantidade de litros $\times$ Preço do litro): `40.0 *. 5.0` $\longrightarrow$ `200.0`

Implementação

```
fn custo_combustivel(distancia, preco_do_litro, rendimento) {
    distancia /. rendimento *. preco_do_litro
}
```

Verificação

Como verificar se a implementação faz o que é esperado?

Executando os exemplos que fizemos anteriormente:

```
> custo_combustivel(400.0, 5.0, 10.0)
200.0
```

2. Depois que você fez o programa para o Alan, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Solução exercício combustível

É possível resolver este problema (produzindo como saída o tipo de combustível) usando as coisas que vimos até aqui? Não!

O que está faltando? Algum tipo de expressão condicional.

Depois voltamos à esse problema!

Condicional

A maioria das linguagens tem **if** para expressar seleção, mas a linguagem Gleam não tem. Ao invés do **if**, ela oferece o **case**, que é mais genérico. Vamos supor por um instante que Gleam tivesse **if**.

A forma geral do **if** poderia ser:

```
if expressão {    // condição
    expressão...
} else {
    expressão...
}
```

Exemplos

```
> if 4 > 2 { 10 + 2 } else { 7 * 3 }
12
> if 10 == 12 { 10 + 2 } else { 7 * 3 }
21
```

Qual a diferente desse **`if`** em relação ao das outras linguagens?

Esse **`if`** é uma expressão, ele produz um valor como resultado. Na maioria das outras linguagens o **`if`** é uma sentença (*statement* em inglês), ele não produz um resultado mas gera um efeito colateral.

Vamos ver como fazer seleção usando o **`case`**.

A forma inicial do **case** é:

```
case expressão {          // expressão examinada
    True -> expressão     // caso True
    False -> expressão    // caso False
}
```

Onde a ordem dos casos não é importante.

```
> if 4 > 2 {
    10 + 2
  } else {
    7 * 3
  }
12
```

```
> case 4 > 2 {
    True -> 10 + 2
    False -> 7 * 3
  }
12
```

Regra de avaliação do `case`

A regra de avaliação de expressões **`case`** é:

- Avalie a expressão examinada
- Se o valor da expressão examinada for `True`, substitua toda a expressão **`case`** pela expressão do caso `True`
- Senão, substitua toda a expressão **`case`** pela expressão do caso `False`

Vamos escrever uma função para calcular o valor absoluto de um número, isto é

$$\mathrm{abs}(x) = \begin{cases} x & \text{se } x \geq 0 \\ -x & \text{caso contrário} \end{cases}$$

e ver o processo de avaliação dessa função.

Exemplo abs

```
fn abs(x) {
  case x >= 0 {
    True -> x
    False -> -x
  }
}
```

```
abs(-4)              // Substitui abs(-4) pelo corpo ...

case -4 >= 0 {       // A expressão axaminada é avaliada
  True -> -4
  False -> - { -4 }
}

case False {         // Como a expressão axaminada é False
  True -> -4         // o case é substituido pela expressão
  False -> - { -4 } // do caso False
}

- { -4 }             // Reduz - { -4 } para 4

4
```

Vamos atualizar a nossa definição de expressão pra incluir o **`case`**.

Definição de expressão (versão 1.0)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva; ou
- Um nome; ou
- Um case; ou
- Uma chamada de função

Regra para **avaliação de expressão**

- Literal $\rightarrow$ valor que o literal representa
- Função primitiva $\rightarrow$ sequência de instruções de máquina associada com a função
- Nome $\rightarrow$ valor associado com o nome no ambiente
- Case $\rightarrow$ avalie usando a regra de avaliação do case
- Chamada de função $\rightarrow$ avalie usando a regra de avaliação de chamadas de funções

Exercício sinal

Defina a função `sinal` que determina o sinal de um número inteiro.

```
> sinal(4)
1
> sinal(0)
0
> sinal(-7)
-1
```

```
fn sinal(x) {
  case x > 0 {
    True -> 1
    False ->
      case x == 0 {
        True -> 0
        False -> -1
      }
  }
}
```

Exercício and

Defina a função **and** que recebe os argumentos booleanos **x** e **y** e produz como resposta o e lógico entre eles, isto é

```
> and(False, False)
False
> and(False, True)
False
> and(True, False)
False
> and(True, True)
True
```

Observe o exemplos e responda: se verificarmos o valor de **x**, e ele for `False`, qual é o resultado da função? `False`.
E se **x** for `True`? Verificamos o valor de **y**, se for `True`, o resultado da função é `True`, senão, o resultado da função é `False`.

```
fn and(x, y) {
  case x {
    False -> False
    True ->
      case y {
        True -> True
        False -> False
      }
  }
}
```

Exercício and

```
fn and(x, y) {
  case x {
    False -> False
    True ->
      case y {
        True -> True
        False -> False
      }
  }
}
```

```
fn and(x, y) {
  case x {
    False -> False
    True -> y
  }
}
```

Podemos simplificar a função? Sim!

Exercício or

Defina a função `or` que recebe os argumentos booleanos `x` e `y` e produz como resposta o ou lógico entre eles, isto é

```
> or(False, False)
False
> or(False, True)
True
> or(True, False)
True
> or(True, True)
True
```

```
fn and(x, y) {
  case x {
    True -> True
    False -> y
  }
}
```

Limitações das funções and e or

Existe alguma implicação em definirmos `and` e `or` como funções?

Sim, elas serão avaliadas como funções, ou seja, todos os argumentos são avaliados antes das funções serem avaliadas e isso impede que algumas otimizações sejam feitas.

Especificamente, na implementação do `and`, se a primeira expressão for `False`, não é necessário avaliar a segunda expressão. De forma semelhante, no `or`, se a primeira expressão for `True`, não é necessário avaliar a segunda expressão.

Essa otimização, chamada de **avaliação em curto-circuito**, é usada em outras linguagens e permitem escrever condições dependentes, como por exemplo `x != 0 and 10 / x == 2`, o que não é possível se todos os argumentos para o `and` são avaliados.

Avaliação em curto-circuito é um tipo de avaliação preguiçosa.

Operadores lógicos

Operadores lógicos

A linguagem Gleam oferece os operadores lógicos `&&` (and), `||` (or) e `!` (not).

Os operadores `&&` e `||` são binários e `!` é unário.

O operador `&&` produz `True` quando os dois operandos são `True`.

O operador `||` produz `True` quando pelo menos um dos dois operandos é `True`.

O operador `!` produz `True` se o operando é `False` e `False` se o operando é `True`.

Exemplos dos operadores lógicos

Alguns exemplos

```
> 3 > 4 || 2 == 1 + 1
True
> 3 > 4 && 2 == 1 + 1
False
```

Diferente do Python, a negação tem prioridade menor que os operadores relacionais.

```
> ! 3 > 4
erro
> !{ 3 > 4 }
True
```

```
> !True
False
> bool.negate(2 == 4)
True
```

O operador && tem maior prioridade do que ||.

```
>>> True || False && False
True
```

```
>>> { True || False } && False
False
```

Avaliação em curto-circuito

Assim como em outras linguagens, os operadores `&&` e `||` são avaliados em curto-circuito.

Ou seja, esses operadores tem regras de avaliação específicas e não são avaliados como funções.

Regra de avaliação da expressão &&

A regra de avaliação da expressão && é:

- Avalie a expressão a esquerda de &&, se o valor for `False`, substitua toda a expressão && por `False`;
- Senão, substitua toda a expressão && pela expressão a direita de &&.

Regra de avaliação da expressão ||

A regra de avaliação da expressão `||` é:

- Avalie a expressão a esquerda de `||`, se o valor for `True`, substitua toda a expressão `||` por `True`;
- Senão, substitua toda a expressão `||` pela expressão a direita de `||`.

Os exemplos a seguir usam o efeito colateral de `io.debug` para demostrar a avaliação em curto-circuito.

```
> 3 > 5 && { io.debug("aqui") True }
False
> 5 > 3 && { io.debug("aqui") True }
"aqui"
True
```

```
> 3 > 5 || { io.debug("aqui") True }
"aqui"
False
> 5 > 3 || { io.debug("aqui") True }
True
```

Igualdade

Igualdade

A linguagem Gleam oferece apenas um operador de igualdade, o `==`, que pode ser usado para quaisquer dois valores do mesmo tipo.

Em Gleam, dois valores são iguais se eles são estruturalmente iguais.

O operador de diferente (negação da igualdade) é `!=`.

Igualdade

```
> 10 == 9 + 1
True
> 3.0 + 1.0 == 4.0
True
> 10 == 10.0
error: Type mismatch
```

```
> ["a", "c", "b"] == ["a", "c", "b"]
True
> [[], [1, 2]] == [[], [1, 2]]
True
> [[], [1, 2]] != [[], [1, 2]]
False
```

Referências

Referências

Básicas

- Tour da linguagem Gleam
- Seção 1.1 - The Elements of Programming (texto mais direto) do livro SICP