Curso de Programação

Evandro Murilo Bronstrup Alves da Silva

8 de maio de 2025

Sumário

Parte I

Expressões

Capítulo 1

Expressões e ordem de precedência

A documentação do PHP diz o seguinte sobre expressões:

> Expressões são os blocos de construção mais importantes do PHP. No PHP, quase tudo o que você escreve são expressões.

É essencial, portanto, saber o que é uma expressão e como ela é interpretada pelo PHP. Esse conhecimento é transferível para qualquer outra linguagem de programação, com alguma adaptação.

O programador escreve instruções que são executadas por uma máquina. Saber expressões é saber como a máquina interpreta essas instruções. O programador que não sabe expressões não sabe o que a máquina faz com o seu código, e logo se perde.

Uma expressão é aquilo que retorna um valor. Para visualizar isso, vamos utilizar um ambiente REPL, isto é, um terminal interativo. Para o PHP usamos o 'psysh' (`https://psysh.org/`). Você digita uma expressão e o 'psysh' retorna o seu valor.

```
> 3+5
= 8

> "o alfa " . "e o ômega"
= "o alfa e o ômega"
```

No primeiro exemplo, o operador '+' é executado e retorna a soma dos seus dois operandos. No seguno exemplo o operador '.' é executado e retorna uma string que é o resultado da concatenação dos seus dois operandos.

Uma ambiguidade pode surgir no caso de uma expressão composta de vários operadores:

```
> 4+2*8
= 20

> "5" * 2 . "3"
= "103"
```

Os operadores são resolvidos em que ordem? No primeiro exemplo, resolve-se primeiro a multiplicação e depois a adição. No segundo exemplo, resolve-se primeiro a multiplicação e depois a concatenação. O nome disso é precedência – os operadores tem prioridade uns sobre os outros, de acordo com as regras estabelecidas pela linguagem de programação. Aqui está uma tabela de precedência dos principais operadores do PHP.

Tabela 1.1: Precedência dos Principais Operadores

Ordem	**Associação**	**Operadores**	**Descrição**
1	Direita	**	Exponenciação
2	Esquerda	* / %	Multiplicação, Divisão, Módulo
3	Esquerda	+ -	Adição, Subtração
4	Esquerda	.	Concatenação de String
5	Não associativo	< <= > >=	Comparação
6	Não associativo	== != === !==	Comparação
7	Esquerda	&&	E lógico
8	Esquerda	||	OU lógico
9	Direita	??	Null coalescing
10	Não associativo	?:	Ternário
11	Direita	= += -= *=	Atribuição

Quanto mais alto na lista, maior a prioridade de execução. Ou seja, a exponenciação tem maior precedência do que a multiplicação. A adição tem maior precedência do que os operadores de comparação. O 'E' lógico tem maior precedência do que o 'OU' lógico, e assim por diante.

Assim como na matemática, parênteses tem precedência máxima, de dentro pra fora.

```
> 2 * 2 + ((5+2) * (1+1))
= 18
```

A resolução do exemplo acima é a seguinte:

```
> 2 * 2 + ((5+2) * (1+1))
= 2 * 2 + (7 * 2)
```

```
= 2 * 2 + 14
= 4 + 14
= 18
```

1. Exercícios guiados

   Cada uma das expressões a seguir deve ser resolvida etapa por etapa, de acordo com a tabela de precedência, de preferência no papel. O objetivo é criar familiaridade com o modelo de execução da linguagem de programação. Confira a resposta apenas após tentar cada exercício.

   Exercício guiado 1.0.1.
   ```
   > 1+1 == 2
   = 2 == 2
   = true
   ```

   Exercício guiado 1.0.2.
   ```
   > 2 == 1+1
   = 2 == 2
   = true
   ```

   Exercício guiado 1.0.3.
   ```
   > (2 == 5-2) == false
   = (2 == 3) == false
   = false == false
   = true
   ```

   Exercício guiado 1.0.4.
   ```
   > 2 + 1 == 3 + 0 * 1
   = 2 + 1 == 3 + 0
   = 2 + 1 == 3
   = 3 == 3
   = true
   ```

   Exercício guiado 1.0.5.
   ```
   > ($a = 2) + 4 * 2 == 6
   = ($a = 2) + 4 * 2 == 6
   = 2 + 4 * 2 == 6
   = 2 + 8 == 6
   = 10 == 6
   = false
   ```

   Exercício guiado 1.0.6.
   ```
   > $a = 2 + 4 * 2
   ```

```
= $a = 2 + 8
= $a = 10
= 10
```

Exercício guiado 1.0.7.

```
> $a + 2 * ($a = 3)
= $a + 2 * 3
= $a + 6
= 3 + 6
= 9
```

1.1 Resolução de variáveis

O último exercício da seção anterior contém um exemplo simples de resolução de variáveis. Vamos analisá-lo agora.

```
> $a + 2 * ($a = 3)
```

Num primeiro momento, essa expressão pode parecer insolúvel: o primeiro elemento dela é a variável '$a', que não está definida. Em PHP, tentar acessar uma variável não definida causa erro:

```
> $b + 2
Undefined variable $b in eval()'d code
```

Acontece que o PHP resolve os elementos da expressão de acordo com a ordem de precedência, então a princípio não precisamos do valor de '$a'. Primeiro olhamos para a subexpressão entre parênteses '($a = 3)'. O resultado desta expressão é '3', com o que chamamos de um efeito colateral: a variável '$a' adquire o valor '3'.

Na sequência, ficamos com '$a + 2 * 3'. Ainda não é a hora de resolver o valor de '$a', já que o operador '*' tem precedência sobre '+'. Resulta disso '$a + 6'. Agora precisamos resolver os elementos da esquerda para a direita. '$a' tem valor '3', do que resulta '3 + 6', finalmente '9'.

1. Exercícios guiados

   Algumas dessas expressões podem resultar em erro.

   Exercício guiado 1.1.1.

   ```
   > $b * ($b = 5) * 2
   = $b * 5 * 2
   = 5 * 5 * 2
   = 10 * 2
   = 20
   ```

Exercício guiado 1.1.2.

```
> 5 + ($a = 2) + ($b = 3) + ($b = 4) + ($a * $b)
= 5 + 2 + ($b = 3) + ($b = 4) + ($a * $b)
= 5 + 2 + 3 + ($b = 4) + ($a * $b)
= 5 + 2 + 3 + 4 + ($a * $b)
= 5 + 2 + 3 + 4 + (2 * 4)
= 5 + 2 + 3 + 4 + 8
= 7 + 3 + 4 + 8
= 10 + 4 + 8
= 14 + 8
= 22
```

Exercício guiado 1.1.3.

```
> ($a = 2 * $b) + ($b = 2)
Undefined variable $b in eval()'d code
```

Exercício guiado 1.1.4.

```
> ($b = 2) + ($a = 2 * $b)
= 2 + ($a = 2 * $b)
= 2 + ($a = 2 * 2)
= 2 + 4
= 6
```

1.2 Escopo

A expressão a seguir faz perfeito sentido matemático.

```
$a + 2 * ($a + 3)
```

Matematicamente, podemos distribuir o 2 na multiplicação, ‘a + 2a + 6‘, e depois juntar as variáveis, ‘3a + 6‘.

Em PHP, esta expressão pode ou não fazer sentido. Se a variável ‘$a‘ estiver definida, a expressão pode ser executada. Caso contrário, temos erro de variável indefinida.

Em outras palavras, a execução de uma expressão que contém variáveis depende delas estarem definidas no contexto de execução. Ou ainda: a resolução de variáveis depende do contexto de execução.

A esse contexto damos o nome de escopo. Podemos entender o escopo como nada mais do que um mapa com as variáveis disponíveis para a execução da expressão. Sempre que usamos o operador de atribuição ‘=‘, estamos alterando o escopo. Vamos representar o escopo da seguinte maneira:

```
{nome: valor, ...}
```

Daqui pra frente podemos representar o escopo após a execução das expressões.

```
> $a = 10;
= 10
> $b = $a + 1;
= 11

{a: 10, b: 11}
```

Também passaremos a omitir o sinal de '$' na resolução das expressões.

```
{a: 5, b: 10}
> 5 + 20 + $a == $b + 12
= 25 + a == b + 12
= 25 + 5 == b + 12
= 30 == b + 12
= 30 == 10 + 12
= 30 == 22
= false
```

1. Exercícios guiados

   Para esses exercícios, lembre-se que a associação do operador de atribuição é da direita para a esquerda, ou seja, é o contrário dos operadores de adição e multiplicação, que vão da esquerda para a direita.

   Exercício guiado 1.2.1.

   ```
   > $b * ($b = $c + ($c = 2 * 3))
   = b * (b = c + (c = 6))
   = b * (b = c + 6) {c: 6}
   = b * (b = 6 + 6)
   = b * (b = 12)
   = b * 12 {c: 6, b: 12}
   = 12 * 12
   = 144
   ```

   Exercício guiado 1.2.2.

   ```
   > $b = $c + $c = 3 * 4
   = b = c + c = 12
   = b = c + 12 {c: 12}
   ```

```
= b = 12 + 12
= b = 24
= 24 {c: 12, b: 24}
```

A etapa crítica do exercício anterior é a resolução de ‘b = c + c = 12‘. Pode não fazer muito sentido a princípio. Já que a precedência de ‘+‘ é maior do que ‘=‘, não deveríamos ter um erro ao tentar resolver ‘c + c‘? Acontece que, neste caso, (c + c) = 12 não faz sentido, porque à esquerda do operador ‘=‘ precisa estar uma variável. Imagine que {c: 2}, então teríamos ‘4 = 12‘, o que sintaticamente não faz sentido no PHP. Nesse caso o interpretador separa a expressão como ‘(b = (c + (c = 12)))‘. Isso é o que chamamos de edge-case, e fica aqui somente como curiosidade.

Exercício guiado 1.2.3.

```
{a: 10, c: 2}
> 400 / $a ** $c * 4
= 400 / a ** 2 * 4
= 400 / 10 ** 2 * 4
= 400 / 100 * 4
= 4 * 4
= 16
```

As variáveis ‘a‘ e ‘c‘ foram resolvidas em etapas separadas para destacar a associação do operador de exponenciação ‘**‘ (direita). No papel, você poderia resolver em uma etapa só.

Exercício guiado 1.2.4.

```
{b: 15, d: 4}
> $b + ($d ** 3 + ($d + $b))
= b + (d ** 3 + (4 + 15))
= b + (d ** 3 + 19)
= b + (4 ** 3 + 19)
= b + (64 + 19)
= b + 83
= 15 + 83
= 98
```

1.3 Funções como expressão

Conforme vimos na introdução, uma expressão é aquilo que retorna um valor. O PHP tem uma sintaxe para definir funções em formato de expressão.

```
fn(parâmetros, ...) => expressão
```

Por exemplo, a função a seguir retorna o valor do parâmetro ‘a‘ somado com 10.

```
fn($a) => $a + 10
```

Como fica isso no REPL?

```
> fn($a) => $a + 10
= Closure($a) {...}
```

O importante é saber que a expressão ‘fn($a) => $a + 10‘ devolveu um valor que representa a função. Para executar a função, usamos parênteses.

```
> (fn($a) => $a + 10)(5)
= 15
```

Dizemos então que aplicamos o valor 5 à função. No nosso modelo de interpretação, podemos assumir que, ao ver uma expressão do tipo ‘(exp)(argumentos)‘, devemos primeiro resolver ‘exp‘ e depois aplicar os ‘argumentos‘ ao resultado. Se o resultado de ‘exp‘ não for uma função, então temos um erro.

```
> (10)(5)
Value of type int is not callable.
```

Vamos olhar novamente para ‘(fn($a) => $a + 10)(5)‘. Podemos enxergar isso da seguinte maneira: estamos aplicando ‘5‘ à função ‘fn($a) => $a + 10‘. Quer dizer que, no contexto de execução da função, ‘$a‘ vai assumir o valor ‘5‘. Podemos definir isso melhor ainda: ao aplicar um argumento à uma função, criamos um escopo onde o valor do argumento é mapeado para o nome do parâmetro. Vamos ver etapa por etapa.

```
> (fn($a) => $a + 10)(5)
{a: 5}
= a + 10
= 5 + 10
= 15
```

Em ‘(exp)(argumentos)‘, se o valor o argumento for em si uma expressão, ela deve ser resolvida primeiro.

```
> (fn($a) => $a + 10)(2*3)
= (fn(a) => a + 10)(6)
{a: 6}
= a + 10
= 6 + 10
= 16
```

1. Exercícios guiados

   Resolva a aplicação de função desses exercícios etapa por etapa, para fixar bem o modelo de interpretação.

   Exercício guiado 1.3.1.

   ```
   > (fn($a, $b) => $a * $b)(4, 9)
   {a: 4, b: 9}
   = a * b
   = 4 * 9
   = 36
   ```

   Exercício guiado 1.3.2.

   ```
   {a: 5}
   > (fn($a, $b) => $a * $b)($a+2, 2)
   = (fn(a, b) => a * b)(5+2, 2)
   = (fn(a, b) => a * b)(7, 2)
   {a: 7, b: 2}
   = a * b
   = 5 * 2
   = 10
   ```

   Exercício guiado 1.3.3.

   ```
   > (fn($a, $b) => $a * $b)(5, 3) + (fn($a, $b) => $a + $b)(3, 5)
   # esquerda
   {a: 5, b: 3}
   = a * b
   = 5 * 3
   = 15

   # direita
   {a: 3, b: 5}
   = a + b
   = 3 + 5
   = 8
   ```

```
# topo
= 15 + 8
= 23
```

Neste exercício nós separamos as aplicações de função em blocos (esquerda, direita). Cada uma com o seu próprio escopo.

Exercício guiado 1.3.4.

```
{a: 7}
> (fn($a) => $a * $a)(3) + $a
# esquerda
{a: 3}
= a * a
= 3 * 3
= 9

# topo
{a: 7}
= 9 + a
= 9 + 7
= 16
```

Neste exercício vemos algo muito importante: o escopo de execução da função não altera o escopo anterior, que aqui chamamos de topo.

Exercício guiado 1.3.5.

```
{a: 3, b: 2}
> (fn($b) => $b * $b)($a) * (fn($a) => $a * $a)($b) + $a
= (fn(b) => b * b)(3) * (fn(a) => a * a)(b) + a
# esquerda
{b: 3}
= b * b
= 3 * 3
= 9

# topo
{a: 3, b: 2}
= 9 * (fn(a) => a * a)($b) + a

# direita
{a: 2}
= a * a
```

```
= 2 * 2
= 4

# topo
{a: 3, b: 2}
= 9 * 4 + a
= 9 * 4 + a
= 36 + a
= 36 + 3
= 39
```

Perceba que abrimos um bloco sempre que conveniente para separar o escopo das funções, com um nome arbitrário (poderíamos ter chamado facilmente "esquerda"de "função A"e "direita"de "função B"). No papel, você pode também fazer um diagrama com setas, ou então separar os escopos em caixas. Sempre que chamamos "topo", retornamos à visão geral.

1.4 Exercícios da primeira parte

Esses exercícios não vão ter resolução passo a passo neste livro, mas você pode facilmente conferir a resposta final executando a expressão inicial no REPL.

Exercício 1.4.1. *Resolva as expressões a seguir passo a passo.*

```
a. 20 / 4 + 5 * 3
b. 3 ** 2 * 4
c. 7 + 2 * 4 / 2 - 7
d. 12 + 12 * 2 + 12 * 3 == 12 * 6
e. 10 + 10 + 10 == 10 * 3
f. 10 * 2 ==  40 / 2
g. 3 + 3 + 3 == 4 + 4 == 8
h. 3 * 2 == 2 * 3 == 6 / 2
```

Exercício 1.4.2. *Resolva as expressões a seguir considerando o escopo {a: 15, c: 5, d: 3}.*

```
a. a / c == d
b. c * d == a
c. c * 3 == a
d. (a + 2 * c) > d * 8
```

```
e. (a = c = d) == 3
f. (a == c) == (d == c)
```

Exercício 1.4.3. *Descreva em uma frase o que cada uma das funções faz. Por exemplo: 'fn(a) => a * 2' pode ser descrita como "calcula o dobro do parâmetro 'a'".*

```
a. fn(a, b) => a + b
b. fn(a) => a * a
c. fn(a) => a + a
d. fn(a) => a - a
e. fn(a, b) => a > b
f. fn(a, b) => a ** b
g. fn(a) => a % 2 == 0
h. fn(a, b) => a % b == 0
i. fn(a) => a < 100
j. fn(a) => a * a * a
```

Exercício 1.4.4. *Resolva as expressões a seguir considerando o escopo {a: 7, b: 3}.*

```
a. (fn(a) => a * a)(b)
b. (fn(a, b) => a + b / 2)(a, b * 2)
c. (fn(a) => a / 2)(6) + a + b
d. (fn(a, b, c) => a + b - c)(a, b, 10) ** 2
e. (fn(a) => a > 5)(a) == (fn(a) => a > 5)(b)
f. (fn(c) => c + 100)(100*3) / 4
```

Capítulo 2

Funções a fundo

Na primeira parte vimos algumas noções sobre funções. Agora é a hora de se aprofundar no assunto, veremos detalhes de escopo, funções compostas, funções recursivas, e outro método para declaração de funções. O objetivo é criar um bom modelo mental de como funciona a execução de funções, e começar a praticar.

2.1 Armazenando funções

Já vimos a seguinte sintaxe para declaração de funções

```
fn(parâmetros, ...) => expressão
```

Pela definição de uma expressão, sabemos que ela retorna um valor. No REPL aparece algo assim

```
> fn($a) => $a % 2 == 0
= Closure($a) {#4140 ...2}
```

'Closure' é o nome da classe interna do PHP que representa este tipo de função. '#4140' é apenas um identificador gerado automaticamente para esta função. Como todo valor, esta função pode ser armazenada numa variável.

```
> $x = fn($a) => $a % 2 == 0
= Closure($a) {#1}
```

Para aplicar esta função a algum valor, usamos parênteses.

```
{x: fn#1}
> $x(5)
= false
> $x(6)
= true
```

Esse valor pode ser passado pra frente.

```
{x: fn#1}
> $y = $x
= Closure($a) {#1}
> $x == $y
= true

{x, y: fn#1}
```

O operador de igualdade '==' no exemplo anterior demonstra como as duas variáveis referenciam exatamente o mesmo valor.

```
> $x = fn($a) => $a % 2 == 0
= Closure($a) {#1}
> $y = fn($a) => $a % 2 == 0
= Closure($a) {#2}
> $x == $y
= false
{x: fn#1, y: fn#2}
```

No exemplo anterior, cada expressão cria uma nova função, com um identificador distinto.

2.2 Funções compostas

Vamos dar um nome melhor para a função usada de exemplo na seção anterior.

```
> $par = fn($a) => $a % 2 == 0
= Closure($a) {#3}
> $par(3)
= false
> $par(8)
= true
```

Isso funciona porque o operador de módulo '%' é equivalente ao resto da divisão. Ou seja, se '$a' for divisível por 2, a sobra vai ser zero, e a função retorna 'true'. Se '$a' não for divisível, a sobra vai ser '1', e a função retorna 'false'.

```
{par: fn#3}
> $par(4)
{a: 4}
= 4 % 2 == 0
= 0 == 0
= true

> $par(3)
= 3 % 2 == 0
= 1 == 0
= false
```

Exercício guiado 2.2.1. *Pense em pelo menos três maneiras de escrever a função 'impar'.*

```
1. fn($a) => $a % 2 != 0
2. fn($a) => $a % 2 == 0
3. fn($a) => !$par
```

A terceira definição de 'impar' é o que chamamos de função composta, isto é, uma função que faz uso de outras funções. Temos ainda uma quarta maneira de definir 'impar', que usaremos no restante desta seção.

```
{par: fn#1}
> $impar = fn($a) => $par($a+1)
= Closure($a) {#2}
```

Essa definição faz uso do fato de que o número subsequente a um número par é sempre ímpar (e vice-versa).

```
{par: fn#1, impar: fn#2}
> $impar(5)
{a: 5}
= par(a+1)
= par(5+1)
= par(6)
```

```
{a: 6}
= a % 2 == 0
= 6 % 2 == 0
= true
```

2.3 Operador ternário

O operador ternário '?:' é um condicional em forma de expressão. Normalmente tem o formato 'expressao$_a$? expressao$_b$: expressao$_c$'. Caso o resultado da 'expressao$_a$' seja 'true', o valor do ternário será 'expressao$_b$', caso contrário, o valor do ternário será 'expressao$_c$'.

```
> $a = 3+2
= 5
> $b = 7-2
= 5
> $a == $b ? "iguais" : "diferentes"
= 5 == 5 ? "iguais" : "diferentes"
= true ? "iguais" : "diferentes"
= "iguais"
```

Exercício guiado 2.3.1. *Escreva uma função que retorne o maior entre dois números, e demonstre a execução.*

```
> $max = fn($a, $b) => $a > $b ? $a : $b
> $max(5, 7)
{a: 5, b: 7}
= a > b ? a : b
= 5 > 7 ? a : b
= false ? a : b
= b
= 7
```

Exercício guiado 2.3.2. *Escreva uma função que aceite uma idade como parâmetro e retorne "maior de idade"ou "menor de idade", se a idade for maior ou menor a 18. Demonstre a execução.*

```
> $idade = fn($a) => $a < 18 ? "menor de idade" : "maior de idade"
> $idade(16)
{a: 16}
= a < 18 ? "menor de idade" : "maior de idade"
= 16 < 18 ? "menor de idade" : "maior de idade"
```

```
= true ? "menor de idade" : "maior de idade"
= "menor de idade"
```

2.4 Escopo revisitado

Vamos começar esta seção com um exercício guiado.

Exercício guiado 2.4.1. *Qual o retorno da última expressão na sequência?*

```
> $par = fn($a) => $a % 2 == 0
> $impar = fn($a) => $par($a+1)
> $par = fn($a) => $a % 2 == 0 ? "sim" : "não"
> $impar(5)
```

O desafio deste exercício consiste num só, que é saber qual versão da função 'par' será executada. A primeira retorna um valor booleano, a segunda retorna uma string. Ou seja, precisamos saber qual será a execução correta:

```
> $impar(5)
{a: 5}
= par(a+1)
= par(5+1)
= par(6)

# Versão A
{a: 6}
= a % 2 == 0
= 6 % 2 == 0
= 0 == 0
= true

# Versão B
{a: 6}
= a % 2 == 0 ? "sim" : "não"
= 6 % 2 == 0 ? "sim" : "não"
= 0 == 0 ? "sim" : "não"
= true ? "sim" : "não"
= "sim"
```

A resposta correta é a 'Versão A'. Para entender o motivo, precisamos aperfeiçoar o nosso entendimento de escopo. Vamos passo a passo, adicionando identificadores para cada uma das funções, e mostrando o escopo principal após cada expressão.

```
{}
> $par = fn($a) => $a % 2 == 0
= Closure($a) {#1}
{par: fn#1}
> $impar = fn($a) => $par($a+1)
= Closure($a) {#2}
{par: fn#1, impar: fn#2}
> $par = fn($a) => $a % 2 == 0 ? "sim" : "não"
= Closure($a) {#3}
{par: fn#3, impar: fn#2}
```

Nesse momento, a primeira função par ‘fn#1‘ não existe mais no escopo principal, tendo sido substituída pela ‘fn#3‘. Como, então, ao aplicar ‘$impar(5)‘, o identificador ‘$par‘ é resolvido como ‘fn#1‘, isto é, como a função ‘$impar‘ encontra a primeira versão da função ‘$par‘?

É preciso saber que, quando uma função é criada da maneira que aprendemos, ela recebe uma cópia do escopo atual. Isto é, a função ‘$impar‘ carrega em seu escopo uma cópia do escopo principal. Vamos tentar representar isso.

```
{par: fn#1}
> $impar = fn($a) => $par($a+1)
= Closure($a) {#2}
{par: fn#1, impar: fn#2{par: fn#1}}
> $par = fn($a) => $a % 2 == 0 ? "sim" : "não"
= Closure($a) {#3}
{par: fn#3, impar: fn#2{par: fn#1}}
```

Essa cópia do escopo, no caso ‘{par: fn#1}‘, não é mais alterada após a criação da função. Quer dizer que o escopo principal e o escopo da função são independentes entre si. Agora fica fácil entender a execução de ‘$impar‘.

```
{par: fn#3, impar: fn#2{par: fn#1}}
> $impar(5)
{par: fn#1, a: 5}
= par(a+1)
= par(6)
{a: 6}
= a % 2 == 0 ? "sim" : "não"
= 6 % 2 == 0 ? "sim" : "não"
= 0 == 0 ? "sim" : "não"
= true ? "sim" : "não"
= "sim"
```

2.5 Exercícios da segunda parte

Exercício 2.5.1. *Escreva uma função para calcular o cubo de um número. Lembre-se, o cubo de um número é o número multiplicado por ele mesmo três vezes.*

Exercício 2.5.2. *Escreva uma função que receba dois números e retorne o menor deles.*

Exercício 2.5.3. *Escreva uma função que receba dois números e retorne o primeiro elevado ao segundo. Dica: dê uma olhada na tabela de operadores, na primeira parte.*

Exercício 2.5.4. *(Desafio) Escreva uma função para multiplicar duas frações.*

Para multiplicar as frações $\frac{2}{3}$ e $\frac{4}{5}$, seguimos os passos:

1. *Multiplicamos os numeradores:* $2 \times 4 = 8$.
2. *Multiplicamos os denominadores:* $3 \times 5 = 15$.
3. *A fração resultante é* $\frac{8}{15}$.

Portanto, $\frac{2}{3} \times \frac{4}{5} = \frac{8}{15}$.

Dica: você pode receber os numeradores e denominadores como parâmetros separados, e retornar o resultado como string, com o operador de concatenação '.'.