

# Fundamentos

## Paradigma de Programação Funcional

Marco A L Barbosa



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

# Conteúdo

Introdução

Combinações

Definições

Modelo de substituição

Condicional

Operadores lógicos

Como projetar funções

Referências

O estudo utilizando apenas este material **não é suficiente** para o entendimento do conteúdo. Recomendamos a leitura das referências no final deste material e a resolução (por parte do aluno) de todos os exercícios indicados.

# Introdução

# Introdução

- ▶ O paradigma de programação funcional é baseado na definição e aplicação de funções
- ▶ As funções são escritas em termos de expressões
- ▶ Mas o que são expressões e como o interpretado avalia uma expressão?

# Expressões

- ▶ Definição de expressão (versão 0.1)
- ▶ Uma expressão consiste de
  - ▶ Um literal; ou
  - ▶ Uma função primitiva

# Tipos primitivas

- ▶ Números
  - ▶ Exatos
    - ▶ Inteiros 1345
    - ▶ Racionais  $1/3$
    - ▶ Complexos com as partes real e imaginária exatas
  - ▶ Inexatos
    - ▶ Ponto flutuante 2.65
    - ▶ Complexos com parte real ou imaginária inexata

# Tipos primitivas

- ▶ Booleano
  - ▶ #t verdadeiro
  - ▶ #f falso
- ▶ Strings
  - ▶ “Seu nome”
- ▶ Muitos outros tipos



# Funções primitivas

- ▶ Aritméticas: +, -, \*, /
- ▶ Relacionais: >, >=, <, <=, =
- ▶ Strings: `string-length`, `string-append`
- ▶ Muitas outras

# Avaliação de expressões

- ▶ Processo de avaliação de expressões (versão 0.1)
  - ▶ Literal → valor que o literal representa
  - ▶ Função primitiva → sequência de instruções de máquina associada com a função
- ▶ Exemplos

```
> #t
```

```
#t
```

```
> 231
```

```
231
```

```
> "Banana"
```

```
"Banana"
```

```
> +
```

```
#<procedure:+>
```

# Combinações

# Combinações

- ▶ Uma expressão que representa uma função pode ser combinada com expressões que representam dados para formar uma expressão que representa a aplicação da função aos dados

```
> (+ 12 56)
```

```
68
```

```
> (* 4 20)
```

```
80
```

```
> (> 4 5)
```

```
#f
```

```
> (string-append "Apenas " "um " "teste")
```

```
"Apenas um teste"
```

- ▶ Este tipo de expressão é chamada de **combinação**

# Combinações

- ▶ Uma combinação consiste de uma lista de expressões entre parênteses
  - ▶ A expressão mais a esquerda é o **operador**
  - ▶ As outras expressões são os **operandos**
- ▶ O valor de uma combinação é obtido aplicando a função especificada pelo operador aos argumentos, que são os valores dos operandos

# Combinações

- ▶ A convenção de colocar o operador a esquerda dos operandos é chamada de **notação pré-fixa**
- ▶ Vantagens da notação pré-fixa

- ▶ Funções podem receber um número variado de argumentos

```
> (* 2 8 10 1)
160
```

- ▶ Combinações podem ser aninhadas facilmente, isto é, os elementos das combinações podem também ser combinações

```
> (+ (* 3 5) (- 10 6) 5)
24
```

```
> (+ (* 3
      (+ (* 2 4)
          (+ 3 5)))
      (+ (- 10 7)
          6))
```

```
57
```

# Expressões

- ▶ Vamos atualizar a definição de expressão para incluir as combinações

# Expressões

- ▶ Vamos atualizar a definição de expressão para incluir as combinações
- ▶ Definição de expressão (versão 0.2)
- ▶ Uma expressão consiste de
  - ▶ Um literal; ou
  - ▶ Uma função primitiva; ou
  - ▶ Uma combinação



# Avaliação de expressões

- ▶ Processo de avaliação de expressões (versão 0.2)
  - ▶ Literal → valor que o literal representa
  - ▶ Função primitiva → sequência de instruções de máquina associada com a função
  - ▶ Combinação
    - ▶ **Avalie cada expressão** da combinação, isto é, reduza cada expressão para um valor
    - ▶ Aplique a função aos argumentos

## Avaliação de expressões

- ▶ Observe que mesmo sendo um procedimento simples, ele pode ser usado para avaliar expressões muito complicadas (como por exemplo, expressões com muitos níveis de aninhamento)
- ▶ Isto é possível porque o procedimento é recursivo
- ▶ A recursão é uma ferramenta muito poderosa, e ela é essencial para a programação funcional

## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6)) ; (* 2 4) \rightarrow 8$

## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6)) ; (* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6)) ; (+ 3 5) \rightarrow 8$

## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$  ;  $(* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))$  ;  $(+ 3 5) \rightarrow 8$   
 $(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))$  ;  $(+ 8 8) \rightarrow 16$

## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$  ;  $(* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))$  ;  $(+ 3 5) \rightarrow 8$   
 $(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))$  ;  $(+ 8 8) \rightarrow 16$   
 $(+ (* 3 16) (+ (- 10 7) 6))$  ;  $(* 3 16) \rightarrow 48$

## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$  ;  $(* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))$  ;  $(+ 3 5) \rightarrow 8$   
 $(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))$  ;  $(+ 8 8) \rightarrow 16$   
 $(+ (* 3 16) (+ (- 10 7) 6))$  ;  $(* 3 16) \rightarrow 48$   
 $(+ 48 (+ (- 10 7) 6))$  ;  $(- 10 7) \rightarrow 3$

## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$  ;  $(* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))$  ;  $(+ 3 5) \rightarrow 8$   
 $(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))$  ;  $(+ 8 8) \rightarrow 16$   
 $(+ (* 3 16) (+ (- 10 7) 6))$  ;  $(* 3 16) \rightarrow 48$   
 $(+ 48 (+ (- 10 7) 6))$  ;  $(- 10 7) \rightarrow 3$   
 $(+ 48 (+ 3 6))$  ;  $(+ 3 6) \rightarrow 9$



## Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$  ;  $(* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))$  ;  $(+ 3 5) \rightarrow 8$   
 $(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))$  ;  $(+ 8 8) \rightarrow 16$   
 $(+ (* 3 16) (+ (- 10 7) 6))$  ;  $(* 3 16) \rightarrow 48$   
 $(+ 48 (+ (- 10 7) 6))$  ;  $(- 10 7) \rightarrow 3$   
 $(+ 48 (+ 3 6))$  ;  $(+ 3 6) \rightarrow 9$   
 $(+ 48 9)$  ;  $(+48 9) \rightarrow 57$

# Avaliação de expressões

$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$  ;  $(* 2 4) \rightarrow 8$   
 $(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))$  ;  $(+ 3 5) \rightarrow 8$   
 $(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))$  ;  $(+ 8 8) \rightarrow 16$   
 $(+ (* 3 16) (+ (- 10 7) 6))$  ;  $(* 3 16) \rightarrow 48$   
 $(+ 48 (+ (- 10 7) 6))$  ;  $(- 10 7) \rightarrow 3$   
 $(+ 48 (+ 3 6))$  ;  $(+ 3 6) \rightarrow 9$   
 $(+ 48 9)$  ;  $(+48 9) \rightarrow 57$

57

## Definições

# Definições

- ▶ Definições servem para dar nome a objetos computacionais, sejam dados ou funções
- ▶ É a forma de abstração mais elementar

# Definições

- ▶ Em Racket, definições são feitas com o `define`

```
(define x 10)
(define y (+ x 24))
```

```
> y
34
```

- ▶ Quando o interpretador encontra uma construção do tipo

```
(define <nome> <exp>)
```

ele associa `<nome>` ao valor obtido pela avaliação de `<exp>`

- ▶ A avaliação de um nome resulta no objeto associado a ele na sua definição
- ▶ A memória que armazena as associações entre nomes e objetos é chamada de **ambiente**

# Definições

- ▶ O procedimento para avaliação de expressão não serve para definições
  - ▶ `(define x 10)` não significa aplicar a função `define` a dois argumentos, um o valor associado a `x` e o outro o valor `10`, o propósito do `define` é justamente associar o valor `10` a `x`
  - ▶ Ou seja, `(define x 10)` não é uma combinação
- ▶ Exceções a regra geral de avaliação de expressões são chamadas de **formas especiais**
- ▶ `define` é uma forma especial
- ▶ Cada forma especial tem a sua própria regra de avaliação
- ▶ O Racket possui poucas formas especiais, isto significa que é possível aprender a sintaxe da linguagem rapidamente

# Definições

- ▶ Também é possível criar definições de novas funções (chamadas de **funções compostas**)
- ▶ A sintaxe geral é

```
(define (<nome> <parametros>) <corpo>)
```

# Definições

- ▶ Exemplos

```
(define (quadrado x)  
  (* x x))
```

```
(define (soma-quadrados a b)  
  (+ (quadrado a) (quadrado b)))
```

```
> (quadrado 5)
```

```
25
```

```
> (quadrado (+ 2 6))
```

```
64
```

```
> (soma-quadrados (+ 2 2) 3)
```

```
25
```

- ▶ Observe que as funções compostas (definidas pelo usuário) são usadas da mesma forma que as funções pré-definidas



## Modelo de substituição

## Modelo de substituição

- ▶ A definição de expressão e o processo de avaliação de expressões devem ser estendidos para comportar formas especiais e o uso de nomes definidos pelo usuário

# Modelo de substituição

- ▶ A definição de expressão e o processo de avaliação de expressões devem ser estendidos para comportar formas especiais e o uso de nomes definidos pelo usuário
- ▶ Definição de expressão (versão 0.3)
- ▶ Uma expressão consiste de
  - ▶ Um literal; ou
  - ▶ Uma função primitiva; ou
  - ▶ Um nome; ou
  - ▶ Uma forma especial; ou
  - ▶ Uma combinação

# Modelo de substituição

- ▶ Processo de avaliação de expressões (versão 0.3)
  - ▶ Literal → valor que o literal representa
  - ▶ Função primitiva → sequência de instruções de máquina associada com a função
  - ▶ Nome → valor associado com o nome no ambiente
  - ▶ Forma especial → usar a regra específica de cada forma especial
  - ▶ Combinação
    - ▶ Avalie cada expressão da combinação, isto é, reduza cada expressão para um valor
    - ▶ Se o operador é uma função composta, avalie o corpo da função composta **substituindo** cada ocorrência do parâmetro formal pelo argumento correspondente
    - ▶ Senão, aplique a função primitiva aos argumentos
- ▶ Esta forma de aplicar funções compostas é chamada de **modelo de substituição**

# Modelo de substituição

```
(define (quadrado x)  
  (* x x))
```

```
(define (soma-quadrados a b)  
  (+ (quadrado a) (quadrado b)))
```

```
(define (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*

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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) ; Subs (soma-quadrados 6 10) pelo corpo
```



# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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) ; Subs (soma-quadrados 6 10) pelo corpo
(+ (quadrado 6) (quadrado 10)) ; Subs (quadrado 6) pelo corpo ...
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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) ; Subs (soma-quadrados 6 10) pelo corpo
(+ (quadrado 6) (quadrado 10)) ; Subs (quadrado 6) pelo corpo ...
(+ (* 6 6) (quadrado 10)) ; Reduz (* 6 6) para 36
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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) ; Subs (soma-quadrados 6 10) pelo corpo
(+ (quadrado 6) (quadrado 10)) ; Subs (quadrado 6) pelo corpo ...
(+ (* 6 6) (quadrado 10)) ; Reduz (* 6 6) para 36
(+ 36 (quadrado 10)) ; Subs (quadrado 10) pelo corpo ...
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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)          ; Subs (soma-quadrados 6 10) pelo corpo  
(+ (quadrado 6) (quadrado 10)) ; Subs (quadrado 6) pelo corpo ...  
(+ (* 6 6) (quadrado 10))      ; Reduz (* 6 6) para 36  
(+ 36 (quadrado 10))           ; Subs (quadrado 10) pelo corpo ...  
(+ 36 (* 10 10))               ; Reduz (* 10 10) para 100
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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)          ; Subs (soma-quadrados 6 10) pelo corpo  
(+ (quadrado 6) (quadrado 10)) ; Subs (quadrado 6) pelo corpo ...  
(+ (* 6 6) (quadrado 10))      ; Reduz (* 6 6) para 36  
(+ 36 (quadrado 10))           ; Subs (quadrado 10) pelo corpo ...  
(+ 36 (* 10 10))               ; Reduz (* 10 10) para 100  
(+ 36 100)                     ; Reduz (+ 36 100) para 136
```

# Modelo de substituição

```
(define (quadrado x)
  (* x x))
```

```
(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
(define (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)          ; Subs (soma-quadrados 6 10) pelo corpo  
(+ (quadrado 6) (quadrado 10)) ; Subs (quadrado 6) pelo corpo ...  
(+ (* 6 6) (quadrado 10))      ; Reduz (* 6 6) para 36  
(+ 36 (quadrado 10))           ; Subs (quadrado 10) pelo corpo ...  
(+ 36 (* 10 10))               ; Reduz (* 10 10) para 100  
(+ 36 100)                     ; Reduz (+ 36 100) para 136
```

```
136
```

# Modelo de substituição

SemNome - DrRacket\*

Ficheiro Editar Ver Linguagem Racket Insert Tabs Ajuda

```
1 ; #lang racket 2
2
3 (define (quadrado x)
4   (* x x))
5
6 (define (soma-quadrados a b)
7   (+ (quadrado a) (quadrado b)))
8
9 (define (f a)
10  (soma-quadrados (+ a 1) (* a 2)))
11
12 (f 5)| 3
```

1 Beginning Student 12:5

4 [Run] [Step Forward]

2 ; #lang racket

3 (f 5)|

## Modelo de substituição

- ▶ 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)) (* (* 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
```

- ▶ Observe que a resposta obtida foi a mesma do método anterior



## Modelo de substituição

- ▶ Este método de avaliação alternativo de primeiro substituir e depois reduzir, é chamado de **avaliação em ordem normal** (que é um tipo de avaliação atrasada)
- ▶ O método de avaliação que primeiro avalia os argumentos e depois aplica a função é chamado de **avaliação em ordem aplicativa**
- ▶ O Racket usa por padrão a avaliação em ordem aplicativa
- ▶ O Haskell usa avaliação em ordem normal

Condicional

# Condicional

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

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

# Condicional

- ▶ A forma especial `cond` é utilizada para especificar funções deste tipo

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [(< x 0) (- x)]))
```

# Condicional

- ▶ A forma especial `cond` é utilizada para especificar funções deste tipo

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [(< x 0) (- x)]))
```

- ▶ Como as duas condições são mutuamente excludentes, podemos usar o `else`

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

# Condicional

- ▶ A forma geral do cond é

```
(cond
  (<p1> <e1>)
  (<p2> <e2>)
  (<p3> <e3>)
  ...
  [(else <en>)])
```

- ▶ Cada par (<p> <e>) é chamado de **cláusula**
- ▶ A primeira expressão de uma cláusula é chamada de **predicado**, isto é, uma expressão cujo o valor é interpretado como verdadeiro ou falso
- ▶ A segunda expressão de uma cláusula é chamada de **consequente**

# Condicional

- ▶ Expressões `cond` são avaliadas da seguinte maneira
  - ▶ Se o primeiro predicado não é um valor, avalie o predicado e o substitua pelo seu valor. Ou seja, substitua todo o `cond` por um novo `cond` onde o primeiro predicado foi substituído pelo seu valor
  - ▶ Se o primeiro predicado é `true` ou `else`, substitua a expressão `cond` inteira pelo primeiro consequente
  - ▶ Se o primeiro predicado é `false`, remove a primeira cláusula. Isto é, substitua o `cond` por um novo `cond` sem a primeira cláusula
  - ▶ Se não tem mais cláusula, sinalize um erro

# Condicional

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

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



# Condicional

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

(abs -4) ; *Substitui (abs -4) pelo corpo ...*

(cond ; *Como o primeiro predicado não é um valor,*  
[(>= -4 0) -4] ; *a expressão (>= -4 0) é avaliada*  
[else (- -4)]) ; *e substituída pelo seu valor*

# Condicional

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

(abs -4) ; *Substitui (abs -4) pelo corpo ...*

```
(cond ; Como o primeiro predicado não é um valor,
  [(>= -4 0) -4] ; a expressão (>= -4 0) é avaliada
  [else (- -4)]) ; e substituída pelo seu valor
```

```
(cond ; Como o primeiro predicado é falso, a primeira
  [false -4] ; cláusula é removida
  [else (- -4)]) ;
```

# Condicional

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

(abs -4) ; Substitui (abs -4) pelo corpo ...

(cond ; Como o primeiro predicado não é um valor,  
[(>= -4 0) -4] ; a expressão (>= -4 0) é avaliada  
[else (- -4)]) ; e substituída pelo seu valor

(cond ; Como o primeiro predicado é falso, a primeira  
[false -4] ; cláusula é removida  
[else (- -4)]) ;

(cond ; Como o primeiro predicado é else,  
[else (- -4)]) ; o cond é substituído pelo primeiro consequente

(- -4) ; Reduz (- -4) para 4

# Condicional

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

(abs -4) ; Substitui (abs -4) pelo corpo ...

(cond ; Como o primeiro predicado não é um valor,  
[(>= -4 0) -4] ; a expressão (>= -4 0) é avaliada  
[else (- -4)]) ; e substituída pelo seu valor

(cond ; Como o primeiro predicado é falso, a primeira  
[false -4] ; cláusula é removida  
[else (- -4)]) ;

(cond ; Como o primeiro predicado é else,  
[else (- -4)]) ; o cond é substituído pelo primeiro consequente

(- -4) ; Reduz (- -4) para 4

# Condicional

- ▶ A forma especial `if` pode ser usada quando existem apenas dois casos
- ▶ A forma geral do `if` é  
`(if <predicado> <consequente> <alternativa>)`
- ▶ Expressões `if` são avaliadas da seguinte maneira
  - ▶ Se o predicado não é um valor, avalie o predicado e o substitua pelo seu valor
  - ▶ Se o predicado é `true`, substitua toda a expressão `if` pelo consequente
  - ▶ Se o predicado é `false`, substitua toda a expressão `if` pela alternativa

# Condicional

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))
```

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

# Condicional

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))
```

`(abs -4)`                   ; *Substitui (abs -4) pelo corpo ...*

`(if (>= -4 0)`           ; *Como o predicado não é um valor,*  
  `-4`                   ; *a expressão (>= -4 0) é avaliada e*  
  `(- -4))`               ; *substituída pelo seu valor*

# Condicional

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))
```

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

```
(if (>= -4 0)      ; Como o predicado não é um valor,
    -4            ; a expressão (>= -4 0) é avaliada e
    (- -4))      ; substituída pelo seu valor
```

```
(if false         ; Como o predicado é false, a expressão if
    -4            ; é substituída pela alternatia
    (- -4))      ;
```



# Condicional

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))
```

(abs -4) ; Substitui (abs -4) pelo corpo ...

(if (>= -4 0) ; Como o predicado não é um valor,  
-4 ; a expressão (>= -4 0) é avaliada e  
(- -4))) ; substituída pelo seu valor

(if false ; Como o predicado é false, a expressão if  
-4 ; é substituída pela alternatia  
(- -4))) ;

(- -4) ; Reduz (- -4) para 4

# Condicional

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))
```

(abs -4) ; Substitui (abs -4) pelo corpo ...

(if (>= -4 0) ; Como o predicado não é um valor,  
-4 ; a expressão (>= -4 0) é avaliada e  
(- -4))) ; substituída pelo seu valor

(if false ; Como o predicado é false, a expressão if  
-4 ; é substituída pela alternatia  
(- -4))) ;

(- -4) ; Reduz (- -4) para 4

4

# Operadores lógicos

# Operadores lógicos

- ▶ Predicados podem ser compostos usando as formas especiais `and` e `or` e a função `not`
- ▶ A função `(not <e>)` produz `#t` quando `<e>` for avaliado para um valor falso, e `#f` caso contrário

```
> (not (> 5 2))
```

```
#f
```

```
> (not (< 5 2))
```

```
#t
```

# Operadores lógicos

- ▶ A forma geral do `and` é:

```
(and <e1> ... <en>)
```

- ▶ Expressões `and` são avaliadas da seguinte maneira
  - ▶ Se não existem expressões, produza `true`
  - ▶ Se a primeira expressão não é um valor, avalie a primeira expressão e a substitua pelo seu valor
  - ▶ Se a primeira expressão é `false`, produza `false`
  - ▶ Se a primeira expressão é `true`, substitua a expressão `and` por uma nova expressão `and` sem a primeira expressão
- ▶ **Observação:** o passo a passo do Racket é um pouco diferente (não elimina os valores `true`)

# Operadores lógicos

```
(and (> 4 2) true (= 3 3)) ; A primeira expressão não é um valor,  
                             ; logo ela é avaliada e substituída pelo  
                             ; seu valor
```

# Operadores lógicos

`(and (> 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(and true true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

# Operadores lógicos

`(and (> 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(and true true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and



# Operadores lógicos

`(and (> 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(and true true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and (= 3 3))` ; Reduz (= 3 3) para true

# Operadores lógicos

`(and (> 4 2) true (= 3 3))` ; *A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor*

`(and true true (= 3 3))` ; *A primeira expressão é true, então  
; ela é removida do and*

`(and true (= 3 3))` ; *A primeira expressão é true, então  
; ela é removida do and*

`(and (= 3 3))` ; *Reduz (= 3 3) para true*

`(and true)` ; *A primeira expressão é true, então  
; ela é removida do and*

# Operadores lógicos

`(and (> 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(and true true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and (= 3 3))` ; Reduz (= 3 3) para true

`(and true)` ; A primeira expressão é true, então  
; ela é removida do and

`(and )` ; Não tem mais expressões, produz true

# Operadores lógicos

`(and (> 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(and true true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and true (= 3 3))` ; A primeira expressão é true, então  
; ela é removida do and

`(and (= 3 3))` ; Reduz (= 3 3) para true

`(and true)` ; A primeira expressão é true, então  
; ela é removida do and

`(and )` ; Não tem mais expressões, produz true

`true`

# Operadores lógicos

- ▶ A forma geral do or é:

(or <e1> ... <en>)

- ▶ Expressões and são avaliadas da seguinte maneira
  - ▶ Se não existem expressões, produza false
  - ▶ Se a primeira expressão não é um valor, avalie a primeira expressão e a substitua pelo seu valor
  - ▶ Se a primeira expressão é true, produza true
  - ▶ Se a primeira expressão é false, substitua a expressão or por uma nova expressão or sem a primeira expressão
- ▶ **Observação:** o passo a passo do Racket é um pouco diferente (não elimina os valores false)

# Operadores lógicos

```
(or (< 4 2) true (= 3 3)) ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor
```

# Operadores lógicos

`(or (< 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(or false true (= 3 3))` ; A primeira expressão é false, então  
; ela é removida do or

# Operadores lógicos

`(or (< 4 2) true (= 3 3))` ; A primeira expressão não é um valor,  
; logo ela é avaliada e substituída pelo  
; seu valor

`(or false true (= 3 3))` ; A primeira expressão é false, então  
; ela é removida do or

`(or true (= 3 3))` ; A primeira expressão é true; produz true  
  
`true`



Como projetar funções

## Como projetar funções

- ▶ Vamos utilizar as receitas de projeto do livro How to Design Programs para escrever funções
- ▶ Estas receitas de projeto permitem o projeto sistemático de funções
- ▶ Este processo pode não ser relevante para problemas simples, mas é essencial para os demais problemas
- ▶ Seja paciente e em breve você verá a utilidade deste processo
- ▶ Vamos treinar com problemas simples, para depois utilizar o processo em outros problemas

# Como projetar funções

1. Assinatura, propósito e cabeçalho
2. Exemplos
3. Template
4. Código do corpo da função
5. Teste e depuração

# Como projetar funções

- ▶ Cada etapa depende da anterior, mas as vezes pode ser necessário mudar a ordem
- ▶ Por exemplo, talvez você faça primeiro os exemplos para entender melhor o problema e poder escrever a assinatura e o propósito
- ▶ As vezes você está escrevendo o corpo e encontra uma nova condição e deve voltar e alterar o propósito e os exemplos
- ▶ Mas você nunca deve escrever o código diretamente

## Como projetar funções

- ▶ Um template contém a estrutura básica que uma função deve ter, independente dos seu detalhes
- ▶ Em muitos casos o template de uma função é determinado pelos tipos de dados dos seus parâmetros. Este tipo de template é chamado de template dirigido por dados
- ▶ Para tipos atômicos (números, string, booleans) usamos o template:

```
(define (fun-for-atomic a)
  (... a))
```

- ▶ Para outros tipos, vamos construir o template com a definição do tipo

## Exemplo 2.1

Defina uma função que calcule o dobro de um dado valor.

## Exemplo 2.1

- ▶ Passo 1: Assinatura, propósito e cabeçalho

```
;; Número -> Número  
;; Produz o dobro de n.  
(define (dobro n) 0)
```

- ▶ Convenções:
  - ▶ Nome do tipo usando CamelCase
  - ▶ Nome da função em minúsculo usando - para separar as palavras

## Exemplo 2.1

### ► Passo 2: Exemplos

```
;; Número -> Número  
;; Produz o dobro de n.
```

```
(check-equal? (dobro 0) 0)  
(check-equal? (dobro 4) 8)  
(check-equal? (dobro -2) -4)))  
  
(define (dobro n) 0)
```



## Exemplo 2.1

- ▶ Passo 3: Template
- ▶ Como o parâmetro é um número, usamos o template para tipos atômicos

```
;; Número -> Número  
;; Produz o dobro de n.  
  
(check-equal? (dobro 0) 0)  
(check-equal? (dobro 4) 8)  
(check-equal? (dobro -2) -4))  
  
;(define (dobro n) 0)  
  
(define (fun-for-atomic a)  
  (... a))
```

## Exemplo 2.1

- ▶ Passo 3: Template
- ▶ Ajustamos os nomes no template para a função que estamos definindo

```
;; Número -> Número  
;; Produz o dobro de n.  
  
(check-equal? (dobro 0) 0)  
(check-equal? (dobro 4) 8)  
(check-equal? (dobro -2) -4))  
  
;(define (dobro n) 0)  
  
(define (dobro n)  
  (... n))
```

## Exemplo 2.1

- ▶ Passo 4: Código do corpo da função
- ▶ Baseado nos passos anteriores, escrevemos o corpo da função

```
;; Número -> Número  
;; Produz o dobro de n.  
  
(check-equal? (dobro 0) 0)  
(check-equal? (dobro 4) 8)  
(check-equal? (dobro -2) -4)))  
  
;(define (dobro n) 0)  
  
(define (dobro n)  
  (* 2 n))
```

## Exemplo 2.1

Programa completo

```
#lang racket

(require rackunit)
(require rackunit/text-ui)

;; Número -> Número
;; Produz o dobro de n.
(define dobro-tests
  (test-suite
   "dobro tests"
   (check-equal? (dobro 0) 0)
   (check-equal? (dobro 4) 8)
   (check-equal? (dobro -2) -4)))

(define (dobro n)
  (* 2 n))

;; continua ...
```

## Exemplo 2.1

```
;; continuação
```

```
;; Teste ... -> Void
```

```
;; Executa um conjunto de testes.
```

```
(define (executa-testes . testes)
```

```
  (run-tests (test-suite "Todos os testes" testes))
```

```
  (void))
```

```
;; Chama a função para executar os testes.
```

```
(executa-testes dobro-tests)
```

## Exemplo 2.1

- ▶ Passo 5: Teste e depuração
- ▶ `ctrl-R` ou `F5` para executar o programa (e os testes)
- ▶ Resultado

```
3 success(es) 0 failure(s) 0 error(s) 3 test(s) run
```

## Exemplo 2.2

Defina uma função que verifique se um número é par.

## Exemplo 2.3

Defina uma função que encontre a maior palavra entre duas palavras dadas.



## Referências

## Referências básicas

- ▶ Vídeos BSL
- ▶ How to Design Functions (Necessário inscrever-se no curso)
- ▶ Vídeos How to Design Functions
- ▶ Introdução rápida ao Racket
- ▶ Capítulos 1 e 2 (2.1 e 2.2) do Guia Racket
- ▶ Seção 1.1 do livro SICP

## Referências complementares

- ▶ Capítulos 1 e 2 do livro TSLP4