Fundamentos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

Introdução

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

- Cada função é uma conjunto de expressões que mapeia valores de entrada para valores de saída.

Mas o que são expressões?

- Uma expressão é uma entidade sintática que quando avaliada 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

Literais de tipos primitivos

Números Exatos

- Inteiros `1345`
- Racionais `1/3`
- Complexos com as partes real e imaginária exatas

Números Inexatos

- Ponto flutuante `2.65`
- Complexos com parte real ou imaginária inexata

Literais de tipos primitivos

Booleano

- `#t` verdadeiro
- `#f` falso

Strings

- `"Seu nome"`

Muitos outros tipos

Funções primitivas

Aritméticas: `+, -, *, /`

Relacionais: `>, >=, <, <=, =`

Strings: **`string-length`**, **`string-append`**, **`number->string`**, **`string->number`**

Muitas outras...

Processo de avaliação de expressões (versão 0.1)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva

Regra para **avaliação de expressão**

- Literal $\longrightarrow$ valor que o literal representa
- Função primitiva $\longrightarrow$ 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

```
> #t
#t
> 231
231
> "Banana"
"Banana"
> +
#<procedure:+>
```

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 em Racket

```
> (+ 12 56)
68
> (> 4 (+ 1 5))
#f
> (string-append "Apenas " "um " "teste")
"Apenas um teste"
```

Baseado nesses exemplos, como podemos definir o que é um combinação?

Primeira tentativa

Uma combinação começa com abre parêntese, seguido de uma função primitiva, seguido de um ou mais **literais**, seguido de fecha parêntese.

Essa definição é adequada? Não!

O exemplo `(> 4 (+ 1 5))` não está de acordo com essa definição!

Segunda tentativa

Uma combinação começa com abre parêntese, seguido de uma função primitiva, seguido de uma ou mais **expressões**, seguido de fecha parêntese.

Vamos usar uma definição mais genérica.

Uma **combinação** consiste de uma lista não vazia de **expressões** entre parênteses

- A expressão mais a esquerda é o **operador** (deve ser avaliada para uma função)
- As outras expressões são os **operandos**

Qual é o valor produzido pela avaliação de uma combinação?

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

Que tipo de notação é essa!? Parece estranha!

- A convenção de colocar o operador a esquerda dos operandos é chamada de **notação prefixa**.
- A forma como isso é expresso no Racket é através de expressões S (sexp). Expressões S são usadas para denotar listas aninhadas (árvores).

Quais as vantagens e desvantagens de usar sexps?

Operadores aritméticos são tratados como as outras funções e podem receber um número variado de argumentos

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

Combinações podem ser aninhadas facilmente, sem preocupações com prioridades das operações

```
> (+ (* 3 5) (- 10 6) 5)
24
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

Vantagens das sexps

Um programa inteiro pode ser representado com uma sequência de sexp e podemos fazer programas que processam outros programas mais facilmente (Racket e outras linguagens são homoicônicas).

Desvantagens das sexps

Diferente da forma que aprendemos...

Pode requerer mais parênteses.

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 (lista não vazia 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
- Combinação
  - **Avalie cada expressão** da combinação, 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 combinação, 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 que expressões de tamanhos arbitrários sejam criadas.
- O processo de avaliação para uma expressão que é uma combinação 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 que expressões de tamanho arbitrário sejam avaliadas.
- 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 Racket (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

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6)) ; (* 2 4) -> 8
(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))       ; (+ 3 5) -> 8
(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))             ; (+ 8 8) -> 16
(+ (* 3 16) (+ (- 10 7) 6))                  ; (* 3 16) -> 48
(+ 48 (+ (- 10 7) 6))                        ; (- 10 7) -> 3
(+ 48 (+ 3 6))                               ; (+ 3 6) -> 9
(+ 48 9)                                     ; (+ 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 o propósito das definições?

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

- É a forma de abstração mais elementar

Em Racket, as definições são feitas com o `define`

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

```
> y
34
```

Como o Racket interpreta um definição?

Quando o interpretador encontra uma construção do tipo

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

ele associa `<nome>` ao valor obtido pela avaliação de `<exp>` (a memória que armazena as associações entre nomes e objetos é chamada de **ambiente**).

Note que uma definição não é uma combinação (expressão) e por isso o procedimento para avaliação de expressão não serve para definições.

- `(define x 10)` não significa aplicar a função `define` a dois argumentos
- O propósito do `define` é associar o valor `10` ao nome `x`
- Ou seja, `(define x 10)` não é uma combinação (expressão)

Dessa forma, os programas em Racket são compostos de duas construções: expressões e definições.

De forma mais precisa, um programa em Racket é formado por uma sequência de definições e expressões.

Como vimos na definição `(define y (+ x 24))`, nomes podem aparecer em expressões, então precisamos atualizar a nossa definição de expressão. Mas antes, vamos ver como definir novas funções.

A sintaxe geral para definição de novas funções (**funções compostas**) é

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

Definição de função

```
(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
```

Observações:

- A forma que uma função é definida é semelhante a forma que ela é chamada: `(quadrado x)` vs `(quadrado 5)`
- As funções compostas (definidas pelo usuário) são usadas da mesma forma que as funções pré-definidas.

Agora precisamos estender a definição de expressões para incluir nomes e alterar a regra de avaliação de expressões para considerar a aplicação 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 combinação (lista não vazia de **expressões** entre parênteses)

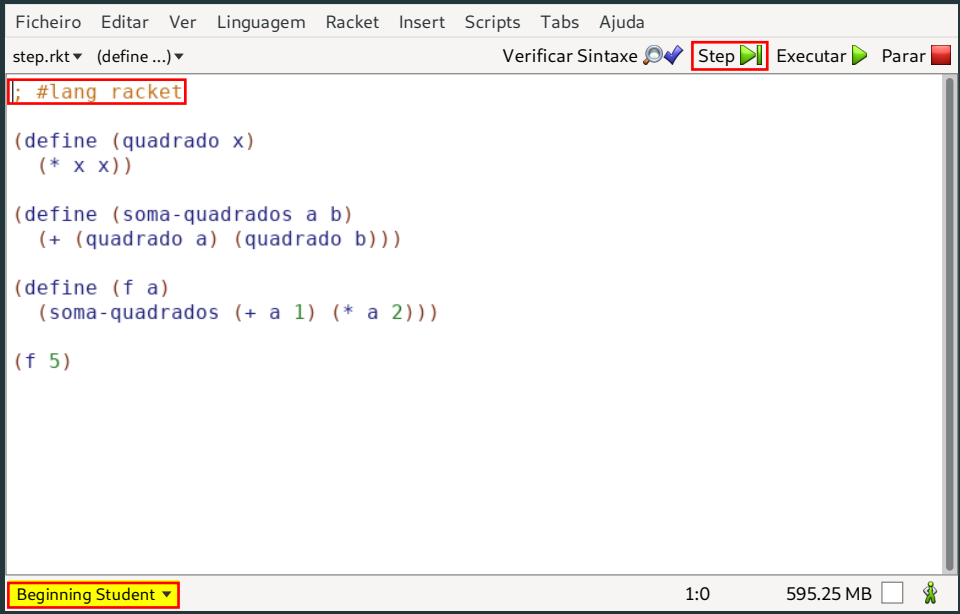
Regra para **avaliação de expressão**

- 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
- Combinação
  - **Avalie cada expressão** da combinação
  - 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 da aplicação de 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

(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
```


Ficheiro Editar Ver Linguagem Racket Insert Scripts Tabs Ajuda
step.rkt (define ...)
Verificar Sintaxe
Step
Executar
Parar
; #lang racket
(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)
Beginning Student
1:0
595.25 MB

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.

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

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.

Exercício

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.

Solução exercício 1

Exemplo de entrada

- Distância: 400 Km
- Preço do litro: R$ 5
- Rendimento: 10 Km/l

Saída

- Quantidade de litros (Distância / Rendimento): $400/10 \longrightarrow 40$
- Custo (Quantidade de litros $\times$ Preço do litro): $40 \times 5 \longrightarrow 200$

Implementação

```
(define (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 5 10)
200
```

Exercício

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 1

É possível resolver este problema (produzindo uma resposta `"alcool"` ou `"gasolina"`) usando as coisas que vimos até aqui? Não!

O que está faltando? Algum tipo de expressão condicional.

Depois voltamos à esse problema!

Condicional

Utilizamos a construção **if** para especificar expressões condicionais. Sua forma geral é

```
(if <predicado> <consequente> <alternativa>)
```

Exemplos

```
> (if (> 4 2) (+ 10 2) (* 7 3))
12
> (if (= 10 12) (+ 10 2) (* 7 3))
21
```

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

O **`if`** do Racket é uma expressão, ele produz um valor como resultado. Na maioria das outras linguagens o **`if`** é uma sentença, ele não produz um resultado mas gera uma mudança no estado do programa.

O **`if`** é uma função? Não.

Se o **`if`** fosse uma função ele seria avaliado usando a regra de avaliação de funções, que diz que todas as expressões dos argumentos da função devem ser avaliados antes da aplicação na função. O **`if`** avalia o consequente ou a alternativa, dependendo da condição, mas não os dois.

O **`if`** é uma **forma especial** e tem uma regra de avaliação específica. (O Racket possui poucas formas especiais, isto significa que é possível aprender a sintaxe da linguagem rapidamente.)

Regra de avaliação do `if`

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 `#t`, substitua toda a expressão **`if`** pelo consequente e avalie o consequente
- Se o predicado é `#f`, substitua toda a expressão **`if`** pela alternativa e avalie a alternativa

Exemplo

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.

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 #f            ; Como o predicado é #f, a expressão if
    -4            ; é substituída pela alternativa
    (- -4))       ;

(- -4)            ; Reduz (- -4) para 4 (não mostrado...)
```

Definição de expressão (versão 1.0)

Vamos atualizar a nossa definição de expressão pra incluir formas especiais.

Definição de expressão (versão 1.0)

Uma **expressão** consiste de

- Um literal; ou
- Uma função primitiva; ou
- Um nome; ou
- Uma forma especial; ou
- Uma combinação

Regra para **avaliação de expressão**

- Literal $\longrightarrow$ valor que o literal representa
- Função primitiva $\longrightarrow$ sequência de instruções de máquina associada com a função
- Nome $\longrightarrow$ valor associado com o nome no ambiente
- Forma especial $\longrightarrow$ avalie a forma especial usando a regra de avaliação específica
- Combinação
  - Avalie cada expressão da combinação
  - 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

A forma especial **cond** pode ser usada quando existem vários (pelo menos um) casos. Por exemplo, podemos utilizar o **cond** ao invés de **if**s na função que determina o sinal de um número.

```
(define (sinal x)
  (if (> x 0)
      1
      (if (= x 0)
          0
          -1)))
```

```
(define (sinal x)
  (cond
    [(> x 0) 1]
    [(= x 0) 0]
    [else -1]))
```

A forma geral do **cond** é

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

Cada par `[<p> <e>]` é chamado de **cláusula** (parênteses e colchetes são equivalentes em Racket).

A primeira expressão de uma cláusula é chamada de **predicado** (expressão cujo o valor é interpretado como verdadeiro ou falso).

A segunda expressão de uma cláusula é chamada de **consequente**.

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 `#t` ou **else**, substitua a expressão **cond** inteira pelo primeiro consequente e avalie o consequente
- Se o primeiro predicado é `#f`, remova 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 (sinal x)
  (cond
    [(> x 0) 1]
    [(= x 0) 0]
    [else -1]))

(sinal 0)     ; Substitui (sinal 0) pelo corpo
              ; corpo da função...

(cond
  [(> 0 0) 1] ; Como o primeiro predicado (> 0 0)
  [(= 0 0) 0] ; não é um valor, ele é avaliado
  [else -1]))

(cond
  [#f 1]      ; Como o primeiro predicado é #f
  [(= 0 0) 0] ; a primeira cláusula é removida
  [else -1]))
```

```
(cond
  [(= 0 0) 0] ; Como o primeiro predicado (= 0 0)
  [else -1])) ; não é um valor, ele é avaliado

(cond
  [#t 0]      ; Como o primeiro predicado é #t
  [else -1])) ; o cond é substituído pelo consequente
              ; da primeira cláusula

0
```

Exercício

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

```
> (e-logico #f #f)
#f
> (e-logico #f #t)
#f
> (e-logico #t #f)
#f
> (e-logico #t #t)
#t
```

Observe o exemplos e responda:

Se verificarmos o valor de `x`, e ele for `#t`, como calculamos o valor da função? Verificando o valor de `y`, se for `#t`, o valor da função é `#t`, senão, o valor da função é `#f`.

E se `x` não for `#t`? O valor da função é `#f`.

```
(define (e-logico)
  (if (equal? x #t)
      (if (equal? y #t)
          #t
          #f)
      #f))
```

Podemos simplificar a função?

```
(define (e-logico)
  (if (equal? x #t)
      (if (equal? y #t)
          #t
          #f)
      #f))
```

Sim!

A expressão `(equal? a #t)` produz o mesmo valor que a expressão `a` (desde que `a` seja booleano). Portanto, podemos escrever

```
(define (e-logico)
  (if x
      (if y #t #f)
      #f))
```

Mais alguma simplificação? Sim, a expressão `(if y #t #f)` é equivalente a expressão `y` (para `y` booleano). Então, podemos escrever

```
(define (e-logico x y)
  (if x y #f))
```

Exercício

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

```
> (ou-logico #f #f)
#f
> (ou-logico #f #t)
#t
> (ou-logico #t #f)
#t
> (ou-logico #t #t)
#t
```

```
(define (ou-logico x y)
  (if x #t y))
```

Existe alguma implicação em definirmos `e-logico` e `ou-logico` 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 `e-logico`, se a primeira expressão for `#f`, não é necessário avaliar a segunda expressão. De forma semelhante, no `ou-logico`, se a primeira expressão for `#t`, 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 e 10 / x == 2`, o que não é possível se todos os argumentos para o `e` são avaliados.

Vamos ver em seguida que o `e` e o `ou` em Racket são formas especiais.

Operadores lógicos

Operadores lógicos

Predicados podem ser compostos usando as formas especiais **`and`** e **`or`** e a função **`not`**

not

A função `(not exp)` produz `#t` quando `exp` é `#f`, e `#f` caso contrário

```
> (not (> 5 2))
#f
> (not (< 5 2))
#t
```

and

A forma geral do **and** é:

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

and

Expressões **and** são avaliadas da seguinte maneira

- Se não existem expressões, produza `#t`
- Se a primeira expressão não é um valor, avalie a primeira expressão e a substitua pelo seu valor
- Se a primeira expressão é `#f`, produza `#f`
- Se a primeira expressão é `#t`, 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 `#t`)

and

```
(and (> 4 2) #t (= 3 3))    ; A primeira expressão não é um valor, logo ela
                            ; é avalida e substiuída pelo seu valor

(and #t #t (= 3 3))         ; A primeira expressão é #t, então
                            ; ela é removida do and

(and #t (= 3 3))            ; A primeira expressão é #t, então ela é removida

(and (= 3 3))               ; Reduz (= 3 3) para #t

(and #t)                    ; A primeira expressão é #t, então ela é removida

(and )                      ; Não tem mais expressões, produz #t

#t
```

or

A forma geral do **or** é:

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

or

Expressões **or** são avaliadas da seguinte maneira

- Se não existem expressões, produza `#f`
- Se a primeira expressão não é um valor, avalie a primeira expressão e a substitua pelo seu valor
- Se a primeira expressão é `#t`, produza `#t`
- Se a primeira expressão é `#f`, 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 `#f`)

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

(or #f #t (= 3 3))       ; A primeira expressão é #f, então
                         ; ela é removida do or

(or #t (= 3 3))          ; A primeira expressão é #t; produz #t

#t
```

Igualdade

Igualdade

Igualdade é o conceito de determinar se dois valores são “iguais”.

É um conceito diferente de igualdade para números (função =, que verifica se dois valores são numericamente iguais).

```
> (= 2 2.0)
#t
```

As principais funções de igualdade em Racket são **`equal?`** e **`eq?`**.

equal?

Cada tipo de objeto determina a sua implementação **equal?**, mas em geral, duas referências são **equal?** se elas referenciam o mesmo objeto, ou se o conteúdo dos objetos são os mesmos.

Duas strings são **equal?** quando elas possuem o mesmo tamanho e contêm a mesma sequência de caracteres

```
> (equal? (substring "banana" 1 4) (substring "cabana" 3 6))
#t
```

equal?

Para estruturas que podem ser compostas, como listas, a função **equal?** checa a igualdade recursivamente

```
> (equal? (list 3 (list 4 2) 5) (list 3 (list 4 2) 5))
#t
> (equal? (list 3 2.0 1) (list 3 2 1))
#f
```

eq?

A função `(eq? v1 v2)` produz `#t` se `v1` e `v2` referenciam o mesmo objeto, `#f` caso contrário. **`eq?`** é avaliada rapidamente pois compara apenas as referências. Entretanto, o **`eq?`** pode não ser adequado, pois a geração dos objetos pode não ser clara

```
> (eq? (+ 2 1) (- 5 2))
#t
> (eq? (expt 2 100) (expt 2 100))
#f
> (eq? (substring "banana" 1 4) (substring "cabana" 3 6))
#f
```

Referências

Referências

Básicas

- Capítulos 1 - Welcome to Racket e 2 - Racket Essentials (2.1 e 2.2) do Guia Racket
- Capítulo 2 - Functions and Programs (texto longo e detalhado) do livro HTDP
- Seção 1.1 - The Elements of Programming (texto mais direto) do livro SICP

Complementares

- Capítulos 1 e 2 do livro TSLP4