Resumo da linguagem Racket

Marco A L Barbosa malbarbo.pro.br

10 de julho de 2024

Conteúdo

1 Tipos de dados e operações primitivas

A seguir apresentamos os tipos e operações primitivas.

1.1 Números

Soma e subtração

```
> (+ 2)
2
> (+ 6 1)
7
> (+ 2 3 4.0)
9.0
> (- 10)
-10
> (- 8 2.0 1)
5.0
```

Multiplicação e divisão

```
> (* 6.0)
6.0
> (* 2 4)
8
> (* 4 2.0 3)
24.0
> (/ 2.0)
0.5
> (/ 42 2 3)
7
```

Exponenciação e radiciação

```
> (sqr 4) ; quadrado
16
> (sqrt 4) ; raiz quadrada
2
> (expt 2 10) ; exponenciação
1024
> (expt 27 1/3) ; raiz cúbica
3.0
```

Divisão inteira e resto da divisão

```
> (quotient 15 6)
2
> (remainder 15 6)
3
```

Desigualdade

```
> (> 3)
#t
> (> 3 5)
#f
> (>= 3 2 1) ; (and (>= 3 2) (>= 2 1))
#t
> (>= 3 2 3) ; (and (>= 3 2) (>= 2 3))
#f
> (<= 4)
#t
> (<= 4 5)
#t
>(< 10 20 40) ; (and (< 10 20) (< 20 40))
#t
```

Máximo e mínimo

```
> (max 10)
10
> (max 5 6 2 4)
6
> (min 8)
8
> (min -1 -2.1)
-2.1
```

Conversão

```
> (inexact->exact 4.0)
4
```

```
> (inexact->exact 2.5)
5/2
> (exact->inexact 3)
3.0
> (number->string 10)
"10"
> (number->string 9.3)
"9.3")
```

Teto, piso e arredondamento

```
> ; teto - menor inteiro maior ou igual
> (ceiling 3.0)
3.0
> (ceiling 3.2)
4.0
> ; piso - menor inteiro menor ou igual
> (floor 3.0)
3.0
> (floor 3.7)
3.0
> (round 4.1)
4.0
> (round 4.5)
5.0
```

Outras funções

```
> (add1 8)
9
> (sub1 7)
6
> (zero? 0)
#t
> (even? 10) ; par
#t
> (odd? 10) ; ímpar
#f
> (positive? 0)
#f
> (positive? 10)
#t
> (negative? -7.2)
#t
> (abs 4)
4
> (abs -12)
12
```

1.2 Strings e caracteres

Quantidade de caracteres

```
> (string-length "Racket")
6
> (string-length "")
0
```

Concatenação

```
> (string-append "Olha o " "gol" "!!!")
"Olha o gol!!!"
> (string-append)
""
> (string-append "aa")
"aa"
```

Substring e divisão

```
> (substring "cadeia de teste" 3 7)
"eia "
> (substring "cadeia de teste" 10)
"teste"
> (string-split "apenas um   teste")
'("apenas" "um" "teste")
> (string-split "   ")
'()
```

Maiúsculas e Minúsculas

```
> (string-downcase "NÃO GRITA @2#1!")
"não grita @2#1!"
> (string-upcase "gol!")
"GOL!"
```

Comparação

```
> (string<=? "casa" "outra")
#t
> (string<=? "casa" "Casa")
#f
> (string<=? "as" "asa")
#t
```

Repetição de caractere

```
> (make-string 3 #\a)
"aaa"
> (make-string 5 #\space)
"     "
```

Conversão para número

```
> (string->number "2.0")
2.0
> (string->number "71")
71
```

1.3 Booleanos

Negação

```
> (not #t)
#f
> (not #f)
#t
```

and

```
> (and #t)
#t
> (and (> 4 3) #t (equal? "casa" "casa"))
```

```
#t
> (and (> 4 3) (= 5 1))
#f
```

or

```
> (or #t)
#t
> (or (= 5 1) #f (> 4 3))
#t
> (or #f (> 5 10) (= 4 1))
#f
```

1.4 Listas

Construção

```
> ; Lista vazia
> empty
'()
> ; Lista com o elemento 3
> (cons 3 empty)
'(3)
> ; Lista com os elementos 1 7 3
> (cons 1 (cons 7 (cons 3 empty)))
'(1 7 3)
> (list 1 7 3)
'(1 7 3)
```

Decomposição

```
> (define lst (cons 1 (cons 7 (cons 3 empty))))
> (first lst)
1
> (rest lst)
'(7 3)
> (first (rest lst))
7
> (second lst)
7
```

Concatenação

```
> (append (list 5 1 2) (list 8 1))
'(5 1 2 8 1)
> (append empty (list 5 4))
'(5 4)
> (append (list 5 8 4) empty)
'(5 8 4)
```

Predicados

```
> (list? empty)
#t
> (list? (cons 3 (cons 2 empty))); ou (list? (list 3 2))
#t
> (empty? empty) ; ou (empty? (list))
#t
> (empty? (cons 3 empty)) ; ou (empty? (list 3))
#f
```

```
> (cons? empty) ; ou (cons? (list))
#f
> (cons? (cons 3 empty)) ; ou (cons? (list 3))
#t
```

Map

```
> (map add1 (list 4 6 10))
'(5 7 11)
> (map list (list 7 2 18))
'((7) (2) (18))
> (map length (list (list 7 2) (list 18) empty))
'(2 1 0)
```

Filter

```
> (filter (negate zero?) (list 4 0 6 0 0 10))
'(4 6 10)
> (filter non-empty-string? (list "casa" "" "rio" ""))
'("casa" "rio")
> (filter cons? (list (list 1 3) empty (list 4) empty))
'((1 3) (4))
```

Fold

```
> (foldr + 0 (list 4 6 10))
20
> (foldr max 30 (list 7 2 18 -20))
30
> (foldr cons empty (list 7 2 18))
'(7 2 18)
```

```
> (foldl + 0 (list 4 6 10))
20
> (foldl max 30 (list 7 2 18 -20))
30
> (foldl cons empty (list 7 2 18))
'(18 2 7)
```

1.5 Igualdade

Números

```
> ; Os valores são númericamente iguais?
> (= 2)
> (= 2 2.0)
#t
> (= 2 2.0 8/4)
#t
```

Outros objetos

```
> ; Referenciam o mesmo objeto?
> (eq? (substring "banana" 1 4) (substring "cabana" 3 6))
#f
> ; O conteúdo dos objetos referenciados são iguais?
> (equal? (substring "banana" 1 4) (substring "cabana" 3 6))
#t
```

2 Definições e formas especiais

Na descrição a seguir `<id>` denota um nome (identificador) qualquer, `expr` uma expressão e `...` repetição da construção.

2.1 Definições

A forma geral para definições é:

```
(define <id> expr)
```

Exemplos

```
> (define x 10)
> (define y (+ x 12))
> y
22
```

As formas gerais para definições de funções são:

```
(define (<id> <id>...)
    expr)
```

Que é equivalente a:

```
(define <id>
  (lambda (<id>...)
    expr)
```

Exemplos

```
> (define (quadrado x)
    (* x x))
> (define soma-quadrados
    (lambda (a b)
      (+ (quadrado a) (quadrado b))))
> (soma-quadrados 3 4)
25
```

2.2 `if`

A forma geral do **`if`** é:

```
(if expr expr expr)
```

A primeira expressão é o predicado, a segunda o consequente e a terceira a alternativa.

Exemplos

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

2.3 `cond`

A forma geral do **`cond`** é

```
(cond
  [expr expr]...
  [else expr])
```

Cada construção `[expr expr]` é chamada de cláusula. A primeira expressão da cláusula é o predicado e a segunda o consequente.

Exemplos

```
> (cond
    [(> 4 2) 10]
    [(= 3 (+ 1 2)) 8]
    [#t 19]
    [else 5])
10
> (cond
    [(< 4 2) 10]
    [(= 3 (+ 1 2)) 8]
    [else 5])
8
> (cond
    [(< 4 2) 10]
    [else 5])
5
```

2.4 `let`

A sintaxe do **`let`** é

```
(let ([var1 exp1]
      [var2 exp2]
      ...
      [varn expn])
  corpo)
```

Os nomes `var1`, `var2`, ..., são locais ao **`let`**, ou seja, são visíveis apenas no corpo do **`let`**.

O resultado da avaliação do `corpo` é o resultado da expressão **`let`**.

No **`let`** os nomes que estão sendo definidos não podem ser usados nas definições dos nomes seguintes, por exemplo, não é possível utilizar o nome `var1` na expressão de `var2`.

`let*` não tem essa limitação.

Exemplos

```
> (let ([a 7]
        [d 8])
    (+ (* a a) d))
57
> (+ (let ([a 10]
           [b 20]) (* a b))
     30)
230
> (let ([x 9])
   (* x
      (let ([x (/ x 3)])
        (+ x x))))
54
> (let ([a 7]
        [b (* 2 a)])
    (add1 b))
; a: undefined; cannot reference an identifier before its definition
> (let* ([a 7]
         [b (* 2 a)])
    (add1 b))
15
```

3 Definição de tipos de dados

Nessa seção mostramos como definir e usar estruturas, enumerações e uniões.

3.1 Estruturas

Uma aproximação da forma geral para definição de estruturas é

```
(struct <nome> (<campo1> ...))
```

Esta construção gera as seguintes funções

```
; Construtor
nome

; Predicado que verifica se um objeto é do tipo da estrutura
nome?

; Seletores
nome-campo1
```

Por exemplos, a definição

```
(struct ponto (x y))
```

Gera as funções `ponto`, `ponto?`, `ponto-x` e `ponto-y`, que podem ser usadas da seguinte forma:

```
> (define p (ponto 3 4))
> (ponto? p) ; p é uma instância de ponto
#t
> (ponto? 23) ; 23 não é uma instância de ponto
#f
> (ponto-x p)
3
> (ponto-y p)
4
> ; Por padrão, o estado interno não é exibido
> p
#<ponto>
> ; Por padrão, a comparação é por referência
> (equal? p (ponto 3 4))
#f
> (equal? p p)
#t
```

Podemos adicionar `#:transparent` a definição de uma estrutura. Isso altera a forma que valores do tipo da estrutura são exibidos e comparados:

```
> (struct ponto (x y) #:transparent)
> (define p (ponto (+ 2 1) 4))
> ; Agora estado interno é exibido
> p
(ponto 3 4)
> ; Agora a compração é pela igualdade dos campos
> (equal? p (ponto 3 4))
#t
> (equal? p p)
#t
```

3.2 Enumerações

Racket não oferece uma forma de declarar enumerações, mas podemos fazer isso em forma de comentários. Por exemplo, para declarar a cor de um semáforo que pode ser verde, amarelo ou vermelho fazemos:

```
;; Cor é um dos valores
;; - "verde"
;; - "amarelo"
;; - "vermelho"
```

3.3 Uniões

Racket não oferece uma forma de declarar uniões, mas podemos fazer isso em forma de comentários. Por exemplo, para declarar o estado de uma tarefa que pode estar em execução, tem sido concluído com sucesso ou com erro, podemos fazer:

```
(struct executando () #:transparent)
;; Representa que uma tarefa está em execução.

(struct sucesso (duracao msg) #:transparent)
;; Representa o estado de uma tarefa que finalizou a execução com sucesso
;; duracao: Número - tempo de execução em segundos
;; msg    : String - mensagem de sucesso gerada pela tarefa

(struct erro (codigo msg) #:transparent)
;; Representa o estado de uma tarefa que finalizou a execução com falha
;; código: Número - o código da falha
;; msg   : String - mensagem de erro gerada pela tarefa

;; EstadoTarefa é um dos valores:
;; - (executando)             A tarefa está em execução
;; - (sucesso Número String)  A tarefa finalizou com sucesso
;; - (erro Número String)     A tarefa finalizou com falha
```

4 Exemplos de programas

Seguem alguns exemplos

4.1 Preço combustível

```
#lang racket

(require examples)

;;;;;;;
;; Tipos de dados

;; Preco é um número positivo.

;; Combustivel é um dos valores
;; - "Alcool"
;; - "Gasolina"

;;;;;;;
;; Funções

;; Preco Preco -> Combustivel
```

```
;;
;; Encontra o combustivel que deve ser utilizado no abastecimento.
;; Produz "Alcool" se preco-alcool menor ou igual a 70% do preco-gasolina,
;; produz "Gasolina" caso contrário.
(examples
 ; (<= preco-alcool preco-gasolina)
 (check-equal? (seleciona-combustivel 3.00 4.00) "Gasolina")
 (check-equal? (seleciona-combustivel 2.90 4.20) "Alcool")
 ; (> preco-alcool preco-gasolina)
 (check-equal? (seleciona-combustivel 3.50 5.00) "Alcool"))

(define (seleciona-combustivel preco-alcool preco-gasolina)
  (if (<= preco-alcool (* 0.7 preco-gasolina))
      "Alcool"
      "Gasolina"))
```

4.2 Ajuste de texto

```
#lang racket

(require examples)

;;;;;;;
;; Tipos de dados

;; Alinhamento é um dos valores
;; - "direita"
;; - "esquerda"
;; - "centro"


;; String Number Alinhamento -> String
;;
;; Produz uma nova string a partir de s que tem exatamente num-chars caracteres
;; e é alinhada de acordo com o alinhamento.
;;
;; Se s tem exatamente num-chars caracteres, então produz s.
;;
;; Se s tem mais do que num-chars caracteres, então s é truncada e ...
;; é adicionado ao final para sinalizar que a string foi abreviada.
;;
;; Se s tem menos do que num-chars caracteres, então espaços são
;; adicionados no início se alinhamento é "esquerda", no fim
;; se alinhamento é "direita", ou no ínicio e fim se alinhamento
;; e "centro". Nesse último caso, se a quantidade de espaços adicionados
;; for impar, então no fim será adicionado 1 espaço a mais do que no início.
(examples
 ; (= (string-length s) num-chars)
 (check-equal? (ajusta-string "casa" 4 "direita") "casa")
 (check-equal? (ajusta-string "casa" 4 "esquerda") "casa")
 (check-equal? (ajusta-string "casa" 4 "centro") "casa")

 ; (> (string-length s) num-chars)
 ; (string-append (substring "casa verde" 0 (- 7 3)) "...")
 (check-equal? (ajusta-string "casa verde" 7 "direita") "casa...")
 (check-equal? (ajusta-string "casa verde" 7 "esquarda") "casa...")
```

```
 (check-equal? (ajusta-string "casa verde" 7 "centro") "casa...")
 (check-equal? (ajusta-string "casa verde" 9 "direita") "casa v...")

 ; (<= (string-length s) num-chars)
 ; direita
 ; (string-append (make-string (- 9 (string-length "casa")) #\space)
 ;                "casa")
 (check-equal? (ajusta-string "casa" 9 "direita") "     casa")

 ; esquerda
 ; (string-append (make-string "casa"
 ;                (- 9 (string-length "casa")) #\space))
 (check-equal? (ajusta-string "casa" 9 "esquerda") "casa     ")

 ; centro
 ; (define num-espacos-inicio (quotient (- num-chars (string-length "casa")) 2)
 ; (define num-espacos-fim (- num-chars (string-length "casa)
 ;                            num-espacos-inicio)
 ; (string-append
 ;   (make-string num-espacos-inicio #\space))
 ;   "centro"
 ;   (make-string num-espacos-fim #\space))
 (check-equal? (ajusta-string "casa" 9 "centro") "  casa   ")
 (check-equal? (ajusta-string "casa" 10 "centro") "   casa   "))

(define (ajusta-string s num-chars alinhamento)
  (cond
    [(= (string-length s) num-chars)
     s]
    [(> (string-length s) num-chars)
     (string-append (substring s 0 (- num-chars 3)) "...")]
    [else
     (define num-espacos (- num-chars (string-length s)))
     (cond
       [(equal? alinhamento "direita")
        (string-append (make-string num-espacos #\space) s)]
       [(equal? alinhamento "esquerda")
        (string-append s (make-string num-espacos #\space))]
       [else
        (define num-espacos-inicio (quotient num-espacos 2))
        (define num-espacos-fim (- num-espacos num-espacos-inicio))
        (string-append
         (make-string num-espacos-inicio #\space)
         s
         (make-string num-espacos-fim #\space))])]))
```

4.3 Máxima repetição

```
;; Lista(Número) -> Número
;;
;; Determina a máxima repetição de lst. Isto é, a maior quantidade
;; de vezes que qualquer elemento de lst se repete.
(examples
 (check-equal? (maxima-repeticao empty) 0)
 (check-equal? (maxima-repeticao (cons 3 empty)) 1)
 (check-equal? (maxima-repeticao (cons 4 (cons 3 empty))) 1)
```

```
 (check-equal? (maxima-repeticao (cons 3 (cons 3 empty))) 2)
 (check-equal? (maxima-repeticao (cons 2 (cons 3 (cons 3 empty)))) 2)
 (check-equal? (maxima-repeticao (cons 3 (cons 2 (cons 3 (cons 3 empty))))) 3))
(define (maxima-repeticao lst)
  (cond
    [(empty? lst) 0]
    [else
     (max
      (numero-de-vezes (first lst) lst)
      (maxima-repeticao (rest lst)))]))

;; Número Lista(Número) -> Número
;;
;; Conta a quantidade de vezes que n aparece em lst.
(examples
 (check-equal? (numero-de-vezes 5 empty) 0)
 (check-equal? (numero-de-vezes 5 (cons 5 empty)) 1)
 (check-equal? (numero-de-vezes 5 (cons 4 empty)) 0)
 (check-equal? (numero-de-vezes 5 (cons 3 (cons 5 empty))) 1)
 (check-equal? (numero-de-vezes 5 (cons 5 (cons 5 empty))) 2)
 (check-equal? (numero-de-vezes 5 (cons 2 (cons 3 (cons 5 empty)))) 1)
 (check-equal? (numero-de-vezes 5 (cons 5 (cons 3 (cons 5 empty)))) 2))

(define (numero-de-vezes n lst)
  (cond
    [(empty? lst) 0]
    [else
     (if (= n (first lst))
         (add1 (numero-de-vezes n (rest lst)))
         (numero-de-vezes n (rest lst)))]))
```