Funções como valores

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

As principais características que vimos até agora do paradigma funcional foram

- Ausência de mudança de estado;
- Tipos algébricos e autorreferências;
- Recursão como forma de especificar iteração.

Veremos a seguir outra característica essencial do paradigma funcional.

Funções como entidades de primeira classe (ou funções como valores)

- Podem ser usadas, sem restrições, onde outros valores podem ser usados (passado como parâmetro, retornado, armazenado em listas, etc);
- Podem ser construídas, sem restrições, onde outros valores também podem (localmente, em expressões, etc);
- Podem ser “tipadas” de forma similar a outro valores, ou seja, existe um tipo associado com cada função e esse tipo podem ser usado para compor outro tipos.

Uma **função de alta ordem** é aquela que recebe como parâmetro uma função ou produz uma função com resultado.

Funções que recebem funções como parâmetro

Funções que recebem funções como parâmetro

Como identificar a necessidade de utilizar funções como parâmetro?

Encontrando similaridades entre funções.

Vamos ver diversas funções e tentar identificar similaridades.

Exemplo: contem-3? e contem-5?

Vamos fazer um exemplo simples. Vamos criar uma função que abstrai o comportamento das funções `contem-3?` e `contem-5?`.

Exemplo: contem-3? e contem-5?

```
;; Lista(Número) -> Boolean
;; Devolve #t se 3 está em lst,
;; #f caso contrário.
(check-equal? (contem-3? (list 4 3 1)) #t)
(define (contem-3? lst)
  (cond
    [(empty? lst) #f]
    [(= 3 (first lst)) #t]
    [else (contem-3? (rest lst))]))
```

```
;; Lista(Número) -> Boolean
;; Devolve #t se 5 está em lst,
;; #f caso contrário.
(check-equal? (contem-5? (list 4 3 1)) #f)
(define (contem-5? lst)
  (cond
    [(empty? lst) #f]
    [(= 5 (first lst)) #t]
    [else (contem-5? (rest lst))]))
```

Vamos definir uma função que abstrai o comportamento de `contem-3?` e `contem-5?`.

```
(define (contem? n lst)
  (cond
    [(empty? lst) #f]
    [(= n (first lst)) #t]
    [else (contem? n (rest lst))]))
```

Exemplo: contem-3? e contem-5?

```
;; Lista(Número) -> Boolean
;; Devolve #t se 3 está em lst,
;; #f caso contrário.
(check-equal? (contem-3? (list 4 3 1)) #t)
(define (contem-3? lst)
  (cond
    [(empty? lst) #f]
    [(= 3 (first lst)) #t]
    [else (contem-3? (rest lst))]))
```

```
;; Lista(Número) -> Boolean
;; Devolve #t se 5 está em lst,
;; #f caso contrário.
(check-equal? (contem-5? (list 4 3 1)) #f)
(define (contem-5? lst)
  (cond
    [(empty? lst) #f]
    [(= 5 (first lst)) #t]
    [else (contem-5? (rest lst))]))
```

Vamos definir uma função que abstrai o comportamento de `contem-3?` e `contem-5?`.

```
(check-equal? (contem? 3 (list 4 3 1)) #t)
(check-equal? (contem? 2 (list 4 3 1)) #f)
(define (contem? n lst)
  (cond
    [(empty? lst) #f]
    [(= n (first lst)) #t]
    [else (contem? n (rest lst))]))
```

Exemplo: contem-3? e contem-5?

```
;; Lista(Número) -> Boolean
;; Devolve #t se 3 está em lst,
;; #f caso contrário.
(check-equal? (contem-3? (list 4 3 1)) #t)
(define (contem-3? lst)
  (cond
    [(empty? lst) #f]
    [(= 3 (first lst)) #t]
    [else (contem-3? (rest lst))]))
```

```
;; Lista(Número) -> Boolean
;; Devolve #t se 5 está em lst,
;; #f caso contrário.
(check-equal? (contem-5? (list 4 3 1)) #f)
(define (contem-5? lst)
  (cond
    [(empty? lst) #f]
    [(= 5 (first lst)) #t]
    [else (contem-5? (rest lst))]))
```

Vamos definir uma função que abstrai o comportamento de `contem-3?` e `contem-5?`.

```
;; Número Lista(Número) -> Boolean
;; Devolve #t se n está em lst,
;; #f caso contrário.
(check-equal? (contem? 3 (list 4 3 1)) #t)
(check-equal? (contem? 2 (list 4 3 1)) #f)
(define (contem? n lst)
  (cond
    [(empty? lst) #f]
    [(= n (first lst)) #t]
    [else (contem? n (rest lst))]))
```

Exemplo: contem-3? e contem-5?

```
;; Lista(Número) -> Boolean
;; Devolve #t se 3 está em lst,
;; #f caso contrário.
(check-equal? (contem-3? (list 4 3 1)) #t)
(define (contem-3? lst)
  (contem? 3 lst))
```

```
;; Lista(Número) -> Boolean
;; Devolve #t se 5 está em lst,
;; #f caso contrário.
(check-equal? (contem-5? (list 4 3 1)) #f)
(define (contem-5? lst)
  (contem? 5 lst))
```

Vamos definir uma função que abstrai o comportamento de `contem-3?` e `contem-5?`.

```
;; Número Lista(Número) -> Boolean
;; Devolve #t se n está em lst,
;; #f caso contrário.
(check-equal? (contem? 3 (list 4 3 1)) #t)
(check-equal? (contem? 2 (list 4 3 1)) #f)
(define (contem? n lst)
  (cond
    [(empty? lst) #f]
    [(= n (first lst)) #t]
    [else (contem? n (rest lst))]))
```

Receita para criar abstração a partir de exemplos

1. Identificar funções com corpo semelhante
    - Identificar o que muda
    - Criar parâmetros para o que muda
    - Copiar o corpo e substituir o que muda pelos parâmetros criados
2. Escrever os exemplos
    - Reutilizar os exemplos das funções existentes
3. Escrever o propósito
4. Escrever a assinatura
5. Reescrever o código da funções iniciais em termos da nova função

Exemplo: `lista-quadrado` e `lista-soma1`

Vamos criar uma função que abstrai o comportamento das funções `lista-quadrado` e `lista-soma1`.

Exemplo: `lista-quadrado` e `lista-soma1`

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; elevado ao quadrado.
(check-equal? (lista-quadrado (list 4)) (list 16))
(define (lista-quadrado lst)
  (cond
    [(empty? lst) empty]
    [else (cons (sqr (first lst))
                (lista-quadrado (rest lst)))]))
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; somado de 1.
(check-equal? (lista-soma1 (list 7 9 1))
              (list 8 10 2)))
(define (lista-soma1 lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (lista-soma1 (rest lst)))]))
```

```
(define (mapeia f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (mapeia f (rest lst)))]))
```

Exemplo: `lista-quadrado` e `lista-soma1`

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; elevado ao quadrado.
(check-equal? (lista-quadrado (list 4)) (list 16))
(define (lista-quadrado lst)
  (cond
    [(empty? lst) empty]
    [else (cons (sqr (first lst))
                (lista-quadrado (rest lst)))]))
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; somado de 1.
(check-equal? (lista-soma1 (list 7 9 1))
              (list 8 10 2)))
(define (lista-soma1 lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (lista-soma1 (rest lst)))]))
```

```
(check-equal? (mapeia sqr (list 4))
              (list 16))
(check-equal? (mapeia add1 (list 7 9 1))
              (list 8 10 2)))
(define (mapeia f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (mapeia f (rest lst)))]))
```

Exemplo: lista-quadrado e lista-soma1

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; elevado ao quadrado.
(check-equal? (lista-quadrado (list 4)) (list 16))
(define (lista-quadrado lst)
  (cond
    [(empty? lst) empty]
    [else (cons (sqr (first lst))
                (lista-quadrado (rest lst)))]))
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; somado de 1.
(check-equal? (lista-soma1 (list 7 9 1))
              (list 8 10 2)))
(define (lista-soma1 lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (lista-soma1 (rest lst)))]))
```

```
;; (Num -> Num) Lista(Num) -> Lista(Num)
;; Devolve uma lista aplicando f a cada
;; elemento de lst, isto é
;; (mapeia f (lista x1 x2 ... xn)) devolve
;; (lista (f x1) (f x2) ... (f xn))
(check-equal? (mapeia sqr (list 4))
              (list 16))
(check-equal? (mapeia add1 (list 7 9 1))
              (list 8 10 2)))
(define (mapeia f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (mapeia f (rest lst)))]))
```

Exemplo: lista-quadrado e lista-soma1

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista cada número de lst
;; elevado ao quadrado.
(check-equal? (lista-quadrado (list 4)) (list 16))
(define (lista-quadrado lst)
    (mapeia sqr lst))
```

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com cada número de lst
;; somado de 1.
(check-equal? (lista-soma1 (list 7 9 1))
              (list 8 10 2)))
(define (lista-soma1 lst)
    (mapeia add1 lst))
```

```
;; (Num -> Num) Lista(Num) -> Lista(Num)
;; Devolve uma lista aplicando f a cada
;; elemento de lst, isto é
;; (mapeia f (lista x1 x2 ... xn)) devolve
;; (lista (f x1) (f x2) ... (f xn))
(check-equal? (mapeia sqr (list 4))
              (list 16))
(check-equal? (mapeia add1 (list 7 9 1))
              (list 8 10 2)))
(define (mapeia f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (mapeia f (rest lst)))]))
```

map

Como resultado do exemplo anterior obtivemos a função **mapeia**, que é pré-definida em Racket com o nome **map**.

```
;; (X -> Y) Lista(X) -> Lista(Y)
;; Devolve uma lista aplicando f a cada elemento de lst,
;; isto é
;; (map f (lista x1 x2 ... xn)) produz
;; (list (f x1) (f x2) ... (f xn))
(define (map f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (map f (rest lst)))]))
```

map - exemplos

```
> (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)
```

Exemplo: `lista-positivos` e `lista-pares`

Vamos criar uma função que abstrai o comportamento das funções `lista-positivos` e `lista-pares`.

Exemplo: `lista-positivos` e `lista-pares`

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores positivos de lst.
(define (lista-positivos lst)
  (cond
    [(empty? lst) empty]
    [(positive? (first lst))
     (cons (first lst)
           (lista-positivos (rest lst)))]
    [else (lista-positivos (rest lst))]))
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores pares de lst.
(define (lista-pares lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (cons (first lst)
           (lista-pares (rest lst)))]
    [else (lista-pares (rest lst))]))
```

```
(define (filtra pred? lst)
  (cond
    [(empty? lst) empty]
    [(pred? (first lst))
     (cons (first lst)
           (filtra pred? (rest lst)))]
    [else (filtra pred? (rest lst))]))
```

Exemplo: lista-positivos e lista-pares

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores positivos de lst.
(define (lista-positivos lst)
  (cond
    [(empty? lst) empty]
    [(positive? (first lst))
     (cons (first lst)
           (lista-positivos (rest lst)))]
    [else (lista-positivos (rest lst))]))
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores pares de lst.
(define (lista-pares lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (cons (first lst)
           (lista-pares (rest lst)))]
    [else (lista-pares (rest lst))]))
```

```
(check-equal? (filtra even? (list 4 2 7)
              (list 4 2)
(check-equal? (filtra positive? (list 3 -2))
              (list 3))
(define (filtra pred? lst)
  (cond
    [(empty? lst) empty]
    [(pred? (first lst))
     (cons (first lst)
           (filtra pred? (rest lst)))]
    [else (filtra pred? (rest lst))]))
```

Exemplo: lista-positivos e lista-pares

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores positivos de lst.
(define (lista-positivos lst)
  (cond
    [(empty? lst) empty]
    [(positive? (first lst))
     (cons (first lst)
           (lista-positivos (rest lst)))]
    [else (lista-positivos (rest lst))]))
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores pares de lst.
(define (lista-pares lst)
  (cond
    [(empty? lst) empty]
    [(even? (first lst))
     (cons (first lst)
           (lista-pares (rest lst)))]
    [else (lista-pares (rest lst))]))
```

```
;; (Num -> Boolean) Lista(Num) -> Lista(Num)
;; Devolve uma lista com todos os elementos
;; x de lst tal que (pred? x) é #t.
(check-equal? (filtra even? (list 4 2 7)
                    (list 4 2)
(check-equal? (filtra positive? (list 3 -2))
                    (list 3))
(define (filtra pred? lst)
  (cond
    [(empty? lst) empty]
    [(pred? (first lst))
     (cons (first lst)
           (filtra pred? (rest lst)))]
    [else (filtra pred? (rest lst))]))
```

Exemplo: `lista-positivos` e `lista-pares`

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores positivos de lst.
(define (lista-positivos lst)
  (filtra positive? lst))
```

```
;; Lista(Número) -> Lista(Número)
;; Devolve uma lista com os valores
;; pares de lst.
(define (lista-pares lst)
  (filtra even? lst))
```

```
;; (Num -> Boolean) Lista(Num) -> Lista(Num)
;; Devolve uma lista com todos os elementos
;; x de lst tal que (pred? x) é #t.
(check-equal? (filtra even? (list 4 2 7)
                           (list 4 2)
(check-equal? (filtra positive? (list 3 -2))
                           (list 3))
(define (filtra pred? lst)
  (cond
    [(empty? lst) empty]
    [(pred? (first lst))
     (cons (first lst)
           (filtra pred? (rest lst)))]
    [else (filtra pred? (rest lst))]))
```

filter

filter

Como resultado do exemplo anterior obtivemos a função `filtra`, que é pré-definida em Racket com o nome `filter`.

```
;; (X -> Boolean) Lista(X) -> Lista(X)
;; Devolve uma lista com todos os elementos de lst
;; tal que pred? é #t.
(define (filter pred? lst)
  (cond
    [(empty? lst) empty]
    [(pred? (first lst))
     (cons (first lst)
           (filter pred? (rest lst)))]
    [else
     (filtra pred? (rest lst))]))
```

`filter` - exemplos

```
> (filter negative? (list 4 6 10))
'()
> (filter even? (list 7 2 18))
'(2 18)
```

Exemplo: soma e produto

Vamos criar uma função que abstrai o comportamento das funções `soma` e `produto`.

Exemplo: soma e produto

```
;; Lista(Número) -> Número
;; Devolve a soma dos números de lst.
(check-equal? (soma (list 4 3 1)) 8)
(define (soma lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (soma (rest lst)))]))
```

```
;; Lista(Número) -> Número
;; Devolve o produto dos números de lst.
(check-equal? (prod (list 4 3 1)) 12)
(define (prod lst)
  (cond
    [(empty? lst) 1]
    [else (* (first lst)
             (prod (rest lst)))]))
```

```
(define (reduz f base lst)
  (cond
    [(empty? lst) base]
    [else (f (first lst)
             (reduz f base (rest lst)))]))
```

Exemplo: soma e produto

```
;; Lista(Número) -> Número
;; Devolve a soma dos números de lst.
(check-equal? (soma (list 4 3 1)) 8)
(define (soma lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (soma (rest lst)))]))

;; Lista(Número) -> Número
;; Devolve o produto dos números de lst.
(check-equal? (prod (list 4 3 1)) 12)
(define (prod lst)
  (cond
    [(empty? lst) 1]
    [else (* (first lst)
             (prod (rest lst)))]))
```

```
(check-equal? (reduz + 0 empty)
              0)
(check-equal? (reduz * 1 (list 3 5 -2))
              -30))
(define (reduz f base lst)
  (cond
    [(empty? lst) base]
    [else (f (first lst)
             (reduz f base (rest lst)))]))
```

Exemplo: soma e produto

```
;; Lista(Número) -> Número
;; Devolve a soma dos números de lst.
(check-equal? (soma (list 4 3 1)) 8)
(define (soma lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (soma (rest lst)))]))

;; Lista(Número) -> Número
;; Devolve o produto dos números de lst.
(check-equal? (prod (list 4 3 1)) 12)
(define (prod lst)
  (cond
    [(empty? lst) 1]
    [else (* (first lst)
             (prod (rest lst)))]))
```

```
;; (Num Num -> Num) Num Lista(Num) -> Num
;; Reduz os valores de lst a um único
;; valor usando a função f.
;; Uma chamada
;; (reduz f base (list x1 x2 ... xn)
;; devolve (f x1 (f x2 ... (f xn base))).
(check-equal? (reduz + 0 empty)
              0)
(check-equal? (reduz * 1 (list 3 5 -2))
              -30))
(define (reduz f base lst)
  (cond
    [(empty? lst) base]
    [else (f (first lst)
             (reduz f base (rest lst)))]))
```

Exemplo: soma e produto

```
;; Lista(Número) -> Número
;; Devolve a soma dos números de lst.
(check-equal? (soma (list 4 3 1)) 8)
(define (soma lst)
  (reduz + 0 lst))
```

```
;; Lista(Número) -> Número
;; Devolve o produto dos números de lst.
(check-equal? (prod (list 4 3 1)) 12)
(define (prod lst)
  (reduz * 1 lst))
```

```
;; (Num Num -> Num) Num Lista(Num) -> Num
;; Reduz os valores de lst a um único
;; valor usando a função f.
;; Uma chamada
;; (reduz f base (list x1 x2 ... xn)
;; devolve (f x1 (f x2 ... (f xn base))).
(check-equal? (reduz + 0 empty)
              0)
(check-equal? (reduz * 1 (list 3 5 -2))
              -30))
(define (reduz f base lst)
  (cond
    [(empty? lst) base]
    [else (f (first lst)
             (reduz f base (rest lst)))]))
```

foldr

foldr

Como resultado do exemplo anterior obtivemos a função **reduz**, que é pré-definida em Racket com o nome **foldr**.

```
;; (X Y -> Y) Y Lista(X) -> Y
;; A chamada
;; (foldr f base (list x1 x2 ... xn) produz
;; (f x1 (f x2 ... (f xn base)))
(define (foldr f base lst)
  (cond
    [(empty? lst) base]
    [else (f (first lst)
             (foldr f base (rest lst)))]))
```

`foldr` - exemplos

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

Funções `map`, `filter` e `foldr`

Quando utilizar as funções `map`, `filter` e `foldr`?

- Quando a lista sempre é processa por inteiro.
- `map`: quando queremos aplicar uma função a cada elemento de uma lista de forma independente.
- `filter`: quando queremos selecionar alguns elementos de uma lista.
- `foldr`: quando queremos calcular um resultado de forma incremental analisando cada elemento de uma lista.

Na dúvida, faça o projeto da função recursiva e depois verifique se ela é um caso específico de `map`, `filter` ou `foldr`.

Exemplo: sinal

Projete uma função que receba como parâmetro uma lista de número e produza uma nova lista com o sinal (1, 0 ou -1) de cada número da lista.

Exemplo: sinal

```
;; Sinal é um dos valores 1, 0, -1

;; Lista(Número) -> Lista(Sinal)
;;
;; Produz uma lista com o sinal de cada
;; elemento de lst.
(examples
  (check-equal? (sinais (list 10 0 2 -4 -1 0 8))
                (list 1 0 1 -1 -1 0 1)))

(define (sinais lst) empty)
```

Podemos usar `map`, `filter` ou `foldr` para implementar a função? Sim, podemos usar o `filter`.

```
(define (sinais lst)
  ;; Número -> Sinal
  ;; Determina o sinal de n.
  (define (sinal n)
    (cond
      [(> n 0) 1]
      [(= n 0) 0]
      [(< n 0) -1]))
  (map sinal lst))
```

Exemplo: pontos nos eixos

Projete uma função que receba como entrada uma lista de pontos no plano cartesiano e indique quais estão sobre o eixo x ou eixo y.

Exemplo: pontos nos eixos

```
(struct ponto (x y) #:transparent)

;; Lista(Ponto) -> Lista(Ponto)
;; Indica quais elementos de pontos estão
;; sobre o eixo x (coordenada y 0) ou
;; eixo y (coordenado x 0).
(examples
  (check-equal? (seleciona-no-eixo
                  (list (ponto 3 0) (ponto 1 3)
                        (ponto 2 0) (ponto 0 2)
                        (ponto 0 0) (ponto 4 7)))
                (list (ponto 3 0) (ponto 2 0)
                      (ponto 0 2) (ponto 0 0))))
(define (seleciona-no-eixo pontos) empty)
```

Podemos usar `map`, `filter` ou `foldr` para implementar a função? Sim, podemos usar o `filter`.

```
(define (seleciona-no-eixo pontos)
  ;; Ponto -> Bool
  ;; Devolve #t se p está sobre o
  ;; eixo x ou y. #f caso contrário.
  (define (no-eixo? p)
    (or (zero? (ponto-x p))
        (zero? (ponto-y p))))
  (filter no-eixo? pontos))
```

Exemplo: ordenação

Projete uma função que receba como entrada uma lista de números e devolva uma lista com os mesmos valores de entrada mas em ordem não decrescente.

Exemplo: ordenação

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Cria uma lista com os elementos de lst
;; em ordem não decrescente.
(examples
 (check-equal? (ordena empty)
               empty)
 (check-equal? (ordena (list 2))
               (list 2))
 (check-equal? (ordena (list 5 2))
               (list 2 5))
 (check-equal? (ordena (list 3 5 2))
               (list 2 3 5))
 (check-equal? (ordena (list 4 3 5 2))
               (list 2 3 4 5)))
(define (ordena lst) empty)
```

Podemos usar `map`, `filter` ou `foldr` para implementar a função? Não está claro... Vamos tentar fazer a implementação usando o modelo.

```
(define (ordena lst)
  (cond
    [(empty? lst) empty]
    [else
      ...
      (first lst)
      (ordena (rest lst))]))
```

Como combinamos o resultado da chamada recursiva com o primeiro elemento? Inserindo o primeiro elemento de forma ordenada.

```
(define (ordena lst)
  (cond
    [(empty? lst) empty]
    [else
      (insere-ordenado (first lst)
                       (ordena (rest lst)))]))
```

Exemplo: ordenação

```
(define (ordena lst)
  (cond
    [(empty? lst) empty]
    [else
      (insere-ordenado (first lst)
                       (ordena (rest lst)))]))
```

E então, podemos usar `map`, `filter` ou `foldr` para implementar a função?

```
(define (foldr f base lst)
  (cond
    [(empty? lst) base]
    [else
      (f (first lst)
         (foldr f base (rest lst)))]))
```

Sim! Podemos usar o `foldr`.

```
(define (ordena lst)
  (foldr insere-ordenado empty lst))
```

Exercício: projete a função `insere-ordenado`.

Exemplos: maiores strings

Projete uma função que receba como entrada uma lista de strings e devolva uma lista com as strings de tamanho máximo entre todas as strings da lista.

Exemplos: maiores strings

```
;; Lista(String) -> Lista(String)
;; Cria uma lista com os elementos de lst que têm tamanho
;; máximo entre todos os elementos de lst.
(examples
  (check-equal? (maiores-strings
                  (list "oi" "casa" "aba" "boi" "eita" "a" "cadê"))
                (list "casa" "eita" "cadê")))

(define (maiores-strings lst) empty)
```

Podemos usar `map`, `filter` ou `foldr` para implementar a função? Não diretamente...

Precisamos separar a solução em duas etapas: encontrar o tamanho máximo e depois selecionar as strings com tamanho máximo.

Exemplos: maiores strings

```
;; Lista(String) -> Número
;;
;; Devolve o tamanho máximo entre todas
;; as strings de lst.
(examples
  (check-equal? (tamanho-maximo
                  (list "oi" "casa" "aba"
                        "boi" "eita" "a"
                        "cadê"))
                4))
```

Podemos usar `map`, `filter` ou `foldr` para implementar a função? Sim, usando o `foldr`, mas parece complicado...

A função para o `foldr` teria que fazer duas coisas, determinar o tamanho de uma string e indicar qual é o máximo entre dois tamanhos.

Podemos separar as etapas de obter os tamanhos e encontrar máximo, usamos o `map` para obter uma lista com os tamanhos e o `foldr` para determinar o valor máximo.

```
(define (tamanho-maximo lst)
  (foldr max 0 (map string-length lst)))
```

Agora podemos implementar a função `maiores-strings`.

Exemplos: maiores strings

```
;; Lista(String) -> Lista(String)
(define (maiores-strings lst)
  (define tmax (tamanho-maximo lst))

  ;; String -> Booleano
  ;; Devolve #t se o tamanho de
  ;; s é igual a tmax. #f caso contrário.
  (define (tamanho-maximo? s)
    (= (string-length s) tmax))

  (filter tamanho-maximo? lst))
```

Existe algo diferente na função `tamanho-maximo?`?

Sim, `tamanho-maximo?` utiliza a variável `tmax`, que não é um parâmetro e nem uma variável local dentro de `tamanho-maximo?`.

Definições locais e fechamentos

Uma **declaração local** é aquela que não é feita no escopo global. As declarações locais, como a de `tmax` e `tamanho-maximo?`, ajudam na escrita e leitura do código e melhoram o encapsulamento.

Uma **variável livre** em relação a uma função é aquela que não é global, não é um parâmetro da função e nem foi declarada localmente dentro da função.

Como uma função acessa um parâmetro ou uma variáveis local? Geralmente, consultando o registro de ativação, o quadro, da sua chamada.

```
;; Lista(String) -> Lista(String)
(define (maiores-strings lst)
  (define tmax (tamanho-maximo lst))

  ;; String -> Booleano
  ;; Devolve #t se o tamanho de
  ;; s é igual a tmax. #f caso contrário.
  (define (tamanho-maximo? s)
    (= (string-length s) tmax))

  (filter tamanho-maximo? lst))
```

Como `tamanho-maximo?` acessa a variável livre `tmax` já que ela não é armazenada no registro de ativação de `tamanho-maximo?`?

A função `tamanho-maximo?` deve “levar” junto com ela a variável livre `tmax`.

O **ambiente léxico** é uma tabela com referências para as variáveis livres.

Um **fechamento** (*closure* em inglês) é uma função junto com o seu ambiente léxico.

```
;; Lista(String) -> Lista(String)
(define (maiores-strings lst)
  (define tmax (tamanho-maximo lst))
  ;; String -> Booleano
  ;; Devolve #t se o tamanho de
  ;; s é igual a tmax. #f caso contrário.
  (define (tamanho-maximo? s)
    (= (string-length s) tmax))
  (filter tamanho-maximo? lst))
```

Quando `tamanho-maximo` é utilizada na chamada do `map` um fechamento é passado como parâmetro.

Até agora, as definições locais que fizemos apareceram no início de uma função, mas as definições também podem aparecer em outros locais.

Considere por exemplo esta função que remove os elementos consecutivos iguais

```
(define (remove-duplicados lst)
  (cond
    [(empty? lst) empty]
    [(empty? (rest lst)) lst]
    [else
     (if (equal? (first lst)
                 (first (remove-duplicados (rest lst))))
         (remove-duplicados (rest lst))
         (cons (first lst)
               (remove-duplicados (rest lst))))]))
```

As expressões `(first lst)` e `(remove-duplicados (rest lst))` são computadas duas vezes.

Podemos utilizar definições locais para armazenar o resultado de expressões e evitar que elas sejam computadas repetidas vezes.

```
(define (remove-duplicados lst)
  (cond
    [(empty? lst) empty]
    [(empty? (rest lst)) lst]
    [else
     (define p (first lst))
     (define r (remove-duplicados (rest lst)))
     (if (equal? p (first r))
         r
         (cons p r))]))
```

O `define` não pode ser usado em alguns lugares, como por exemplo no consequente ou alternativa do **`if`**.

Em geral utilizamos `define` apenas no início da função, em outros lugares utilizamos a forma especial **`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

Definições internas com o **let**

```
(define (remove-duplicados lst)
  (cond
    [(empty? lst) empty]
    [(empty? (rest lst)) lst]
    [else
     (let ([p (first lst)]
           [r (remove-duplicados (rest lst))])
       (if (equal? p (first r))
           r
           (cons p r)))]))
```

Exemplo

Defina a função `mapeia` em termos da função `reduz`.

`mapeia` em termos de `reduz`

```
(define (mapeia f lst)
  (define (cons-f e lst)
    (cons (f e) lst))
  (reduz cons-f empty lst))
```

Exemplo

Defina a função `filtra` em termos da função `reduz`.

`filtra` em termos de `reduz`

```
(define (filtra pred? lst)
  (define (cons-if e lst)
    (if (pred? e) (cons e lst) lst))
  (reduz cons-if empty lst))
```

Funções anônimas

Da mesma forma que podemos utilizar expressões aritméticas sem precisar nomeá-las, também podemos utilizar expressões que resultam em funções sem precisar nomeá-las

Quando fazemos um `define` de uma função, estamos especificando duas coisas: **a função** e **o nome da função**. Quando escrevemos

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

O Racket interpreta como

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

O que deixa claro a distinção entre criar a função e dar nome à função. Às vezes é útil definir uma função sem dar nome a ela.

lambda é a palavra chave usada para especificar funções. A sintaxe do **lambda** é

```
(lambda (parametros ...)
    corpo)
```

Em vez de utilizar a palavra **lambda**, podemos utilizar a letra λ (ctrl + \no DrRacket)

Como e quando utilizar uma função anônima?

- Como parâmetro, quando a função for pequena e necessária apenas naquele local

```
> (map (λ (x) (* x 2)) (list 3 8 -6))
'(6 16 -12)
> (filter (λ (x) (< x 10)) (list 3 20 -4 50))
'(3 -4)
```

- Como resultado de função

Funções que produzem funções

Como identificar a necessidade de criar e utilizar funções que produzem funções?

- Parametrizar a criação de funções fixando alguns parâmetros
- Composição de funções
- ...
- Requer experiência

Exemplo: somador

Defina uma função que receba um parâmetro *n* e devolva uma função que soma o seu argumento a *n*.

Exemplo: somador

```
> (define soma1 (somador 1))
> (define soma5 (somador 5))
> (soma1 4)
5
> (soma5 9)
14
> (soma1 6)
7
> (soma5 3)
8
```

Exemplo: somador

```
;; Número -> (Número -> Número)
;; Devolve uma função que recebe um parâmetro x
;; e produz a soma de n e x.
(examples
 (check-equal? ((somador 4) 3) 7)
 (check-equal? ((somador -2) 8) 6))
(define (somador n) ...)

;; Versão com função nomeada.
(define (somador n)
  (define (soma x)
    (+ n x))
  soma)
```

```
;; Número -> (Número -> Número)
;; Devolve uma função que recebe um parâmetro x
;; e produz a soma de n e x.
(examples
 (check-equal? ((somador 4) 3) 7)
 (check-equal? ((somador -2) 8) 6))
(define (somador n) ...)

;; Versão com função anônima.
(define (somador n)
  (λ (x) (+ n x)))
```

Exemplo: negação

Defina uma função que receba como parâmetro um predicado (função que retorna booleano) e retorne uma função que retorna a negação do predicado.

- `negate` (referência)

Exemplo: negação

```
> ((nega positive?) 3)
#f
> ((nega positive?) -3)
#t
> ((nega even?) 4)
#f
> ((nega even?) 3)
#t
```

Exemplo: negação

```
;; (X -> Boolean) -> (X -> Boolean)
;; Devolve uma função que é semelhante a pred,
;; mas que devolve a  negação do resultado de pred.
;; Veja a função pré-definida negate.
(examples
 (check-equal? ((nega positive?) 3) #f)
 (check-equal? ((nega positive?) -3) #t)
 (check-equal? ((nega even?) 4) #f)
 (check-equal? ((nega even?) 3) #t))

(define (nega pred?)
  (λ (x) (not (pred? x))))
```

Currying

Currying

No cálculo lambda o currying permite definir funções que admitem múltiplos parâmetros.

Aqui o currying permite a aplicação parcial das funções.

Por exemplo, para uma função que admite dois argumentos, poderemos aplicá-la apenas ao primeiro argumento e mais tarde ao segundo argumento, resultando no valor esperado.

Currying

```
> (define f (λ (x) (λ (y) (* x y))))
> (define ((g x) y) (< x y))
> (map (f 2) (list 1 2 3 4))
'(2 4 6 8)
> (filter (g 2) (list 1 2 3 4))
'(3 4)
```

curry e curryr

As funções pré-definidas `curry` e `curryr` são utilizadas para fixar argumentos de funções

- `curry` fixa os argumentos da esquerda para direita
- `curryr` fixa os argumentos da direita para esquerda

curry e curryr

```
> (define e-4? (curry = 4))
> (e-4? 4)
#t
> (e-4? 5)
#f
> (filter e-4? (list 3 4 7 4 6))
'(4 4)
> (filter (curry < 3) (list 4 3 2 5 7 1))
'(4 5 7)
> (filter (curryr < 3) (list 4 3 2 5 7 1))
'(2 1)
> (map (curry + 5) (list 3 6 2))
'(8 11 7)
```

Outras funções de alta ordem

Outras funções de alta ordem

- `apply` (referência)

```
> (apply < (list 4 5))
#t
> (apply + (list 4 5))
9
> (apply * (list 2 3 4))
24
```

- `andmap` (referência)
- `ormap` (referência)
- `build-list` (referência)

Funções com número variado de parâmetros

Muitas funções pré-definidas aceitam um número variado de parâmetros. Como criar funções com esta característica?

Forma geral

```
(define (nome obrigatorios . opcionais) corpo)
(define (nome . opcionais) corpo)
(λ (obrigatorios . opcionais) corpo)
(λ opcionais corpo)
```

Os parâmetros opcionais são agrupados em uma lista

Exemplos

```
> (define (f1 p1 p2 . outros) outros)
> (f1 4 5 7 -2 5)
'(7 -2 5)
> (f1 4 5)
'()
> (f1 4)
f1: arity mismatch;
 the expected number of arguments does not match the given number
  expected: at least 2
  given: 1
  arguments...:
   4
```

Referências

Básicas

- Vídeos Abstractions
- Texto “From Examples” do curso Introduction to Systematic Program Design - Part 1 (Necessário inscrever-se no curso)
- Capítulos 14 e 15 do livro HTDP
- Seções 3.9 e 3.17 da Referência Racket

Complementares

- Seções 1.3 (1.3.1 e 1.3.2) e 2.2.3 do livro SICP
- Seções 4.2 e 5.5 do livro TSPL4