

Funções

Paradigma de Programação Funcional

Marco A L Barbosa



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

<http://github.com/malbarbo/na-func>

Conteúdo

Introdução

Funções que recebem funções como parâmetro

foldr

map

filter

Criando abstração a partir de exemplos

Definições locais e fechamentos

Funções anônimas

Funções que devolvem funções

curry e curryr

Outras funções de alta ordem

Funções com número variado de parâmetros

Introdução

Introdução

- ▶ As duas principais características que vimos até agora do paradigma funcional são
 - ▶ Ausência de mudança de estado
 - ▶ Recursão como forma de especificar iteração
- ▶ Veremos mais uma característica essencial do paradigma funcional
 - ▶ Funções como entidades de primeira classe
 - ▶ Funções como parâmetros
 - ▶ Funções como resultado de uma expressão
 - ▶ Armazenamento de funções em variáveis e estruturas

Introdução

- ▶ Vamos ver um caso simples de criação de abstração a partir de exemplos

Exemplo 1

Vamos criar uma função que abstraia o comportamento das funções `contem-5?` e `contem-3?`.

Funções que recebem funções como parâmetro

Funções que recebem funções como parâmetro

- ▶ Uma **função de alta** ordem é aquela que recebe como parâmetro uma função ou devolve uma função com resultado
- ▶ Como identificar a necessidade de utilizar funções como parâmetro?
 - ▶ Encontrando similaridades em funções
 - ▶ Vamos rever diversas funções que escrevemos e tentar identificar similaridades

Exemplo 2

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

foldr

foldr

Como resultado do exemplo 2 obtivemos a função `reduz`, que é pré-definida em Racket com o nome `foldr`.

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

Exemplos de uso da função `foldr`

```
> (foldr + 0 (list 4 6 10))
20
> (foldr cons empty (list 7 2 18))
'(7 2 8)
> (foldr max 7 (list 7 2 18 -20))
18
```

Exemplo 3

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

map

map

Como resultado do exemplo 3 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
;; (mapeia f (lista x1 x2 ... xn)) devolve
;; (list (f x1) (f x2) ... (f xn))
(define (mapeia f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                 (mapeia f (rest lst)))]))
```

Exemplos de uso da função 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)
```

Exemplo 4

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

filter

filter

Como resultado do exemplo 4 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 verdadeiro.  
(define (filtra pred? lst)  
  (cond  
    [(empty? lst) empty]  
    [else  
     (cond  
       [(pred? (first lst))  
        (cons (first lst) (filtra pred (rest lst)))]  
       [else (filtra pred? (rest lst))])])])
```

Exemplos de uso da função filter

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

Criando abstração a partir de exemplos

Criando abstração a partir de exemplos

O processo que usamos nestes exemplos, foi o mesmo (veja Abstraction from examples para detalhes)

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 testes
 - ▶ reutilizar os testes das funções existentes
3. Escrever o propósito
4. Escrever a assinatura
5. Reescrever o código da funções iniciais em termos da nova função

Definições locais e fechamentos

Definições locais e fechamentos

- ▶ Considere as seguintes definições

```
(define (soma x) (+ x 5))
```

```
(define (lista-soma5 lst)  
  (map soma lst))
```

- ▶ Existem dois problemas com estas definições
 - ▶ A função `soma` tem um uso bastante restrito (supomos que ela é utilizada apenas pela função `lista-soma5`), mas foi declarada em um escopo global utilizando um nome fácil de ter conflito (outro programador pode escolher o nome `soma` para outro função)
 - ▶ A função `lista-soma5` é bastante específica e pode ser generalizada

Definições locais e fechamentos

- ▶ O primeiro problema pode ser resolvido colocando a definição de soma dentro da função lista-soma5, desta forma a função soma é visível apenas para lista-soma5. Isto melhora o encapsulamento e libera o nome soma

```
(define (lista-soma5 lst)
  (define (soma x)
    (+ x 5))
  (map soma lst))
```

Definições locais e fechamentos

- ▶ O segundo problema pode ser resolver adicionado um parâmetro `n` e mudando o nome da função `lista-soma5` para `lista-soma-n`

```
(define (lista-soma-n n lst)
  (define (soma x)
    (+ x n))
  (map soma lst))
```

- ▶ Observe que `soma` utiliza a variável livre `n`

Definições locais e fechamentos

- ▶ Uma **variável livre** é aquela que não foi declarada localmente (dentro da função) e não é um parâmetro
- ▶ Como `soma` pode ser usado fora do contexto que ela foi declarada (como quando ela for executada dentro da função `map`), `soma` deve “levar” junto com ela o ambiente de referenciamento
- ▶ **Ambiente de referenciamento** é uma tabela com as referências para as variáveis livres
- ▶ Um **fechamento** (*closure* em inglês) é uma função junto com o seu ambiente de referenciamento
- ▶ Neste caso, quando `soma` é utilizada na chamada do `map` um fechamento é passado como parâmetro

Definições locais e fechamentos

- ▶ Definições internas também são usadas para evitar computar a mesma expressão mais que uma vez
- ▶ Exemplo de 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))))]))
```

- ▶ Observe que as expressões `(first lst)` e `(remove-duplicados (rest lst))` são computadas duas vezes

Definições locais e fechamentos

- ▶ Criando definições internas obtemos

```
(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))]))
```

- ▶ Desta forma as expressões são computadas apenas uma vez
- ▶ O `define` não pode ser usado em alguns lugares, como por exemplo no conseqüente (ou alternativa) do `if`
- ▶ Em geral utilizamos `define` apenas no início da função, em outros lugares utilizamos o `let`

Definições locais e fechamentos

- ▶ A forma geral 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 esta limitação

Definições locais e fechamentos

► 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 5

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 6

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

Funções anônimas

- ▶ Da mesma forma que podemos utilizar expressões aritméticas sem precisar nomeá-las, também podemos utilizar expressões com funções sem precisar nomeá-las
- ▶ Quando fazemos um `define` de uma função, estamos especificando duas coisas: a função é 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 a função. As vezes é útil definir uma função sem dar nome a ela

Funções anônimas

- ▶ lambda é a forma especial usada para especificar funções. A sintaxe do lambda é
 - ▶ (lambda (parametros ...) corpo)
- ▶ Ao invés de utilizar a palavra lambda, podemos utiliza a letra λ (ctrl-\ no DrRacket)
- ▶ Mas como e quando utilizar uma função anônima?
 - ▶ Como parâmetro, quando a função for pequena e necessária apenas naquele local
 - ▶ Como resultado de função
- ▶ Exemplos

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

Funções que devolvem funções

Funções que devolvem funções

- ▶ Como identificar a necessidade de utilizar funções como resultado?
 - ▶ Parametrizar a criação de funções fixando alguns parâmetros
 - ▶ Composição de funções
 - ▶ ...
 - ▶ Requer experiência

Exemplo 7

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

Exemplo 7

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

Resultado exemplo 7

```
;; Número -> (Número -> Número)
;; Devolve uma função que recebe uma parâmetro x e faz a soma de n e x.
(define somador-tests
  (test-suite
   "somador tests"
   (check-equal? ((somador 4) 3) 7)
   (check-equal? ((somador -2) 8) 6)))

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

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


Exemplo 8

Defina uma função que recebe como parâmetro um predicado (função que retorna verdadeiro ou falso) e retorne uma função que retorna a negação do predicado.

Exemplo 8

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

Resultado exemplo 8

```
;; (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.
(define nega-tests
  (test-suite
   "nega tests"
   (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))))
```

curry e curryr

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
- ▶ Exemplos

```
> (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)
```

Exemplo 9

Defina uma função que implemente o algoritmo de ordenação quicksort.

Quicksort

```
;; Lista(Número) -> Lista(Número)  
;; Ordena una lista de números usando o quicksort.
```

```
(define quicksort-tests  
  (test-suite  
    "quicksort tests"  
    (check-equal? (quicksort empty)  
                  empty)  
    (check-equal? (quicksort (list 3))  
                  (list 3))  
    (check-equal? (quicksort (list 10 3 -4 5 9))  
                  (list -4 3 5 9 10))  
    (check-equal? (quicksort (list 3 10 3 0 5 0 9))  
                  (list 0 0 3 3 5 9 10))))  
  
(define (quicksort lst)  
  (if (empty? lst)  
      empty  
      (let ([pivo (first lst)]  
            [resto (rest lst)])  
        (append (quicksort (filter (curryr < pivo) resto))  
                (list pivo)  
                (quicksort (filter (curryr >= pivo) resto))))))
```

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

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

- ▶ Os parâmetros opcionais são agrupados em uma lista

Funções com número variado de parâmetros

► 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
```