

Dados compostos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.
<http://github.com/malbarbo/na-progfun>

Introdução

- Os tipos de dados que vimos até agora são atômicos, isto é, não podem ser decompostos
- Estamos interessados em representar dados onde dois ou mais valores devem ficar juntos
 - Registro de um aluno
 - Placar de um jogo de futebol
 - Informações de um produto
- Chamamos estes tipos de dados de **estruturas**

- Em Racket utilizamos a forma especial `struct` para definir estruturas
- Vamos definir uma estrutura para representar um ponto em um plano cartesiano

Estruturas

```
(struct ponto (x y))  
(define p1 (ponto 3 4))  
(define p2 (ponto 8 2))
```

```
> (ponto-x p1)
```

```
3
```

```
> (ponto-y p1)
```

```
4
```

```
> (ponto-x p2)
```

```
8
```

```
> (ponto-y p2)
```

```
2
```

```
> (ponto? p1)
```

```
#t
```

```
> (ponto? "ola")
```

```
#f
```

- Uma aproximação da sintaxe do struct é

```
(struct <id-estrutura> (<id-campo-1> ...))
```

Funções definidas na criação de uma estrutura

- Funções definidas pelo struct

;; Construtor

id-estrutura

;; Predicado que teste se o valor é do tipo definido

id-estrutura?

;; Seletores

id-estrutura-id-campo

Funções definidas na criação de uma estrutura

- A estrutura

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

- Define as funções

```
;; Construtor
```

```
ponto
```

```
;; Predicado
```

```
ponto?
```

```
;; Seletores
```

```
ponto-x
```

```
ponto-y
```


- Por padrão, ao exibir um dado estruturado o interpretador não exibe os campos do dado (para preservar o encapsulamento)

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

```
> (ponto 3 4)
```

```
#<ponto>
```

- Podemos usar a palavra chave `#:transparent` para tornar a estrutura “transparente”

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

; mesmo formato de criação e de exibição

```
> (ponto 3 4)
```

```
(ponto 3 4)
```

Estruturas transparentes e a função `equal`?

- Além de mudar a forma que o ponto é exibido, a palavra chave `#:transparent` também altera o funcionamento da função `equal`?

Estruturas transparentes e a função equal?

```
;; Por padrão, dois pontos são iguais se eles são  
;; o mesmo ponto
```

```
(struct ponto (x y))  
(define p1 (ponto 3 4))  
(define p2 (ponto 3 4))
```

```
> (equal? p1 p2)
```

```
#f
```

```
> (equal? p1 p1)
```

```
#t
```

Estruturas transparentes e a função equal?

```
;; Com #:transparent, dois pontos são iguais se os seus  
;; campos são iguais
```

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

```
(define p1 (ponto 3 4))
```

```
(define p2 (ponto 3 4))
```

```
> (equal? p1 p2)
```

```
#t
```

```
> (equal? p1 p1)
```

```
#t
```

- Junto com a definição de uma estrutura, também faremos
 - A descrição do propósito e campos da estrutura
 - Exemplos
 - Modelo

Definindo estruturas

```
(struct ponto (x y))  
;; Ponto representa um ponto no plano cartesiano  
;; x : Número - a coordenada x  
;; y : Número - a coordenada y  
;; Exemplos  
#;  
(define p1 (ponto 3 4))  
#;  
(define p2 (ponto 8 2))
```

Definindo estruturas

```
;; Modelo
```

```
#;
```

```
(define (fun-for-ponto p)  
  (... (ponto-x p)  
        (ponto-y p)))
```

*;; A sequência #; comenta a próxima s-exp. Nos exemplos
;; do primeiro (até o) correspondente.*

Alterando dados estruturados

- Se quisermos mudar um campo de um dado estruturado, temos que criar uma cópia com o campo alterado
- Vamos criar um ponto p2 que é como p1, mas com o valor 5 para o campo y

```
> (define p1 (ponto 3 4))
```

```
> (define p2 (ponto (ponto-x p1) 5))
```

```
> p2
```

```
(ponto 3 5)
```

- Este método é limitado
 - Se a estrutura tem muitos campos e desejamos alterar apenas um campo, temos que especificar a cópia de todos os outros
 - Se a estrutura é alterada, todas as operações de “cópia” devem ser alteradas

- Racket oferece a forma especial `struct-copy` (referência), que facilita este tipo de operação

Alterando dados estruturados

```
> (define p2 (struct-copy ponto p1  
                [y 5]))
```

```
> p2  
(ponto 3 5)
```

```
> (define p3 (struct-copy ponto p2  
                [x 4]))
```

```
> p3  
(ponto 4 5)
```

```
> (define p4 (struct-copy ponto p2  
                [y 9]  
                [x 6]))
```

```
> p4  
(ponto 6 9)
```

Exemplo 3.1

Defina uma função que calcule a distância de um ponto a origem.

Exemplo 3.2

Defina uma estrutura para representar um retângulo. Em seguida defina uma função que classifique um retângulo em largo (largura maior que altura), alto (altura maior que largura) ou quadrado (altura igual a largura).

Estruturas com auto-referência

Estruturas com auto-referência

- Por enquanto estamos trabalhando com quantidades pré-determinadas de dados
- Como representar e processar uma quantidade de dados indeterminada?
 - Vamos criar estruturas com auto-referência, isto é, estruturas em que pelo menos um campo tem o mesmo tipo da estrutura sendo definida
 - Vamos usar funções recursivas para processar estruturas com auto-referência

Listas

- A estrutura recursiva mais comum nas linguagens funcionais é a lista
- Vamos tentar criar uma definição para lista

- A ideia é criar uma estrutura com dois campos. O primeiro campo representa um valor na lista e o segundo campo representa o restante da lista (que é uma lista)

```
(struct lista (primeiro rest) #:transparent)
;; Representa uma lista
;; primeiro: Qualquer - é o primeiro elemento
;;                               da lista
;; rest:      Lista     - é o restante da lista
```

- Utilizando esta definição, vamos tentar representar uma lista com os valores 4, 2 e 8

```
(define lst (lista 4 (lista 2 (lista 8 ???))))
```

- O problema com esta definição é que ela não tem fim. Uma lista é definida em termos de outra lista, que é definida em termos de outra lista, etc

- Precisamos de uma maneira de representar uma lista vazia (sem elementos), desta forma podemos usar a lista vazia para terminar uma sequência de elementos. Em outras palavras, para terminar a definição recursiva, precisamos de uma caso base

- Uma **Lista** é
 - `nil`; ou
 - `(no primeiro rest)` onde `primeiro` é o primeiro elemento da lista e `rest` é uma **Lista** com o restante dos elementos

Listas

```
(define nil (void))

(struct no (primeiro rest) #:transparent)
;; Uma Lista é
;;   - nil; ou
;;   - (no primeiro rest) onde primeiro é o primeiro elemento
;;     da lista e rest é uma Lista com o restante dos elementos
;; Exemplos
#;
(define lst-vazia nil)
#;
(define lst1 (no 3 nil))
#;
(define lst2 (no 10 (no 3 nil)))
#;
(define lst3 (no 1 lst2))
```

```
;; Modelo  
#;  
(define (fun-for-lista lst)  
  (cond  
    [(equal? lst nil) ...]  
    [else ... (no-primeiro lst)  
              ... (fun-for-lista (no-rest lst)) ... ]))
```


- Observe que o modelo foi escrito baseado na definição de Lista
 - A definição tem dois casos, o modelo também
 - A definição é recursiva, o modelo também

Listas

```
> (define lst1 (no 3 nil))           ;Lista com o elemento 3
> (define lst2 (no 8 (no 7 nil)))    ;Lista com os elementos 8 e 7
> lst1
(no 3 #<void>)
> lst2
(no 8 (no 7 #<void>))
> (no-primeiro lst2)
8
> (no-rest lst2)
(no 7 #<void>)
> (no-rest lst1)
#<void>
> (no-primeiro (no-rest lst1))
. . no-primeiro: contract violation
  expected: no?
  given: #<void>
```

Lista

```
;; Lista com os elementos 8 e 7
> (define lst2 (no 8 (no 7 nil)))
;; Defini uma lista a partir de uma lista existente
> (define lst3 (no 4 lst2))
> lst3
(no 4 (no 8 (no 7 #<void>)))
> (no-primeiro lst3)
4
> (no-rest lst3)
(no 8 (no 7 #<void>))
> (no-primeiro (no-rest lst3))
8
```

Exemplo 3.3

Defina uma função que conte a quantidade de elementos de uma lista.

- O Racket já vem com listas pré-definidas
 - `empty` ao invés de `nil`
 - `cons` ao invés de `no`
 - `first` ao invés de `no-primeiro`
 - `rest` ao invés de `no-rest`
- Outras funções
 - `empty?` verifica se uma lista é vazia
 - `list?` verifica se um valor é uma lista

- Lista pré-definida em Racket
- Uma **Lista** é
 - `empty`; ou
 - `(cons first rest)` onde `first` é o primeiro elemento da lista e `rest` é uma **Lista** com o restante dos elementos

```
;; Modelo  
#;  
(define (fun-for-list lst)  
  (cond  
    [(empty? lst) ...]  
    [else ... (first lst)  
              ... (fun-for-list (rest lst)) ... ]))
```

Listas

```
> (define lst1 (cons 3 empty)) ; Lista com o elemento 3
> (define lst2 (cons 8 (cons 7 empty))) ; Lista com 8 e 7
> lst1
'(3)
> lst2
'(8 7)
> (first lst2)
8
> (rest lst2)
'(7)
> (rest (rest lst2))
'()
> (first (rest lst1))
. . first: contract violation
  expected: (and/c list? (not/c empty?))
  given: '()
```


Listas

```
;; Lista com os elementos 8 e 7
> (define lst2 (cons 8 (cons 7 empty)))
;; Defini uma lista a partir de uma lista existente
> (define lst3 (cons 4 lst2))
> lst3
'(4 8 7)
> (first lst3)
4
> (rest lst3)
'(8 7)
> (first (rest lst3))
8
```

- O Racket oferece uma forma conveniente de criar listas

```
> (list 4 5 6 -2 20)
'(4 5 6 -2 20)
```

- Em geral

```
(list <a1> <a2> ... <an>)
```

é equivalente a

```
(cons <a1>
      (cons <a2>
            (cons ...
                  (cons <an> empty) ...))))
```

Exemplo 3.4

Defina uma função que some os valores de uma lista de números.

Exemplo 3.5

Defina uma função que receba dois parâmetros, um valor a e uma lista lst e crie uma nova lista a partir de lst sem a primeira ocorrência de a .

Listas aninhadas

Listas aninhadas

- Às vezes é necessário criar uma lista, que contenha outras listas, e estas listas contenham outras listas, etc

- Exemplo

```
> (list 1 4 (list 5 empty (list 2) 9) 10)
'(1 4 (5 () (2) 9) 10)
```

- Chamamos este tipo de lista de lista aninhada
- Como podemos definir uma lista aninhada?

- Uma **Lista aninhada** é
 - `empty`; ou
 - `(cons lst1 lst2)`, onde `lst1` e `lst2` são **Listas aninhadas**;
ou
 - `(cons a lst)`, onde `a` é um valor que não seja uma Lista aninhada e `lst` é uma **Lista aninhada**

Listas aninhadas

```
;; Modelo
```

```
##;
```

```
(define (fun-for-lista-aninhada lst)
```

```
  (cond
```

```
    [(empty? lst) ...]
```

```
    [(list? (first lst))
```

```
     ... (fun-for-lista-aninhada (first lst))
```

```
     ... (fun-for-lista-aninhada (rest lst)) ...]
```

```
  [else
```

```
   ... (first lst)
```

```
   ... (fun-for-lista-aninhada (rest lst)) ... ]))
```


Exemplo 3.6

Defina uma função que some todos os números de uma lista aninhada de números.

Exemplo 3.7

Defina uma função que aplaine uma lista aninhada, isto é, transforme uma lista aninhada em uma lista sem listas aninhadas com os mesmos elementos e na mesma ordem da lista aninhada.

Árvores binárias

- Como podemos definir uma árvore binária?
- Uma **Árvore binária** é
 - empty; ou
 - (arvore-bin v esq dir), onde v é o valor armazenado no nó e esq e dir são **Árvores binárias**

Árvores binárias

```
;; Modelo
#;
(define (fun-for-arvore-bin t)
  (cond
    [(empty? t) ...]
    [else
     ... (arvore-bin-v t)
     ... (fun-for-arvore-bin (arvore-bin-dir t))
     ... (fun-for-arvore-bin (arvore-bin-esq t)) ...]))
```

Exemplo 3.8

Defina uma função que calcule a altura de uma árvore binária. A altura de uma árvore binária é a distância entre a raiz e o seu descendente mais afastado. Uma árvore com um único nó tem altura 0.

Referências

- Vídeos Compound Data
- Vídeos Self-Reference
- Vídeos Reference
- Capítulos 8 a 12 do livro HTDP
- Seções 2.3, 2.4, 3.8 e 5.1 do Guia Racket

Referências complementares

- Seções 2.1 (2.1.1 - 2.1.3) e 2.2 (2.2.1) do livro SICP
- Seções 3.9 e 4.1 da Referência Racket
- Seção 6.3 do livro TSPL4