Dados compostos

Paradigma de Programação Funcional

Marco A L Barbosa

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

Conteúdo

Pares

Pares

- Da mesma forma que podemos criar funções compostas, também podemos criar dados compostos
- Uma forma de composição de dados em Racket é o par

Pares

- Um par é uma combinação de 2 valores
- Um par é criado com a função primitiva `cons`
- O primeiro valor de um par é acessado com a função `car`
- O segundo valor de um par é acessado com a função `cdr`

```
> (define p (cons 10 20))
> (car p)
10
> (cdr p)
20
```

- Os predicados `pair?` ou `cons?` podem ser usados para testar se um valor é um par

```
> (pair? (cons 1 2))
#t
> (pair? 10)
#f
```

Pares

- Conceitualmente, o par é a única construção necessária para criar qualquer dado composto
- Da mesma forma que uma função composta pode ser usada na criação de outra função composta, um par pode ser usado na criação de outro par
- Por exemplo

```
> (define a (cons (cons 2 4) 6))
> (car a)
'(2 . 4)
> (cdr a)
6
> (car (car a))
2
> (cdr (car a))
4
```

Estruturas

Estruturas

- Queremos criar uma estrutura para representar um ponto cartesiano, com as seguintes funções

```
> (define p (ponto 4 5))
> (ponto-x p)
4
> (ponto-y p)
5
> (ponto? p)
#t
> (ponto? (cons 4 5))
#f
> p
(ponto 4 5)
```

- Como podemos usar pares para fazer isto?

Estruturas

```
(define (ponto x y)
  (cons "ponto" (cons x y)))

(define (ponto? p)
  (and (pair? p)
       (string=? (car p) "ponto")))

(define (ponto-x p)
  (car (cdr p)))

(define (ponto-y p)
  (cdr (cdr p)))

; Como definir a função de impressão?
```

Estruturas

- Embora o par seja suficiente para criar estruturas, ele não é muito conveniente
- O Racket oferece a forma especial `struct` para facilitar a criação de estruturas
- Uma aproximação da sintaxe do `struct` é
  - `(struct <id-estrutura> (<id-campo-1> ...))`
- Funções criadas pelo `struct`
  - `id-estrutura`: construtor
  - `id-estrutura?`: um predicado que testa se o valor é do tipo definido
  - `id-estrutura-id-campo`: uma função de acesso para cada campo

Estruturas

```
> (struct ponto (x y))
> (define p (ponto 3 4))
> p ; os detalhes não são exibidos
#<ponto>
> (ponto-x p)
3
> (ponto-y p)
4
> (ponto? p)
#t
> (ponto? 10)
#f
> (ponto? (cons 3 4))
#f

> (struct ponto (x y) #:transparent)
> (define p (ponto 3 4))
> p ; os detalhes são exibidos por causa do #:transparent
(ponto 3 4)
```

Estruturas

- Se quisermos mudar um atributo de um objeto, temos que criar uma cópia com o atributo alterado
- Vamos criar um ponto p2 que é como p1 mas deslocado 2 unidade no eixo y

```
> (define p1 (ponto 3 4))
> (define p2 (ponto (ponto-x p1) (+ (ponto-y p1) 2)))
> p2
(ponto 3 6)
```

- Este método é um pouco limitado, pois se a estrutura tem muitos campos e desejamos alterar apenas um campo, temos que especificar a cópia de todos os outros
- Racket oferece a forma especial `struct-copy` (referência), que facilita este tipo de operação

```
> (define p2 (struct-copy ponto p1 (y (+ (ponto-y p1) 2))))
> p2
(ponto 3 6)
```

Atenção

Nesta disciplina não vamos usar pares com o propósito de definir estruturas.

Para definir estruturas usaremos ‘struct‘.

Listas

Listas

- Também podemos usar pares para criar lista de valores
- Como representar uma lista sem nenhum elemento?
  - `null` (valor especial pré-definido)
- Como representar uma lista com o elemento 7?
  - `(cons 7 null)`
- Como representar uma lista com os elementos 7 e 14?
  - `(cons 7 (cons 14 null))`
- Como acessar os elementos de uma lista?

```
> (define lst (cons 4 (cons 7 (cons 2 null))))
> (car lst)
4
> (car (cdr lst))
7
> (car (cdr (cdr lst))
2
> (cdr (cdr (cdr lst)))
'()
```

Listas

- 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> null) ...)))
```

Listas

- Quando usamos pares para criar listas, é conveniente utilizar outros nomes para `null`, `car` e `cdr`
  - `empty` ao invés de `null` (lista vazia)
  - `first` ao invés de `car` (primeiro da lista)
  - `rest` ao invés de `cdr` (resto da lista)
- Diferenças
  - `null` e `empty` são equivalente
  - `first` e `rest` só podem ser usadas para listas

```
> (first (cons 1 2))
first: contract violation
expected: (and/c list? (not/c empty?))
given: '(1 . 2)
> (rest (cons 1 2))
rest: contract violation
expected: (and/c list? (not/c empty?))
given: '(1 . 2)
```

Atenção

Não vamos mais usar as funções ‘car‘ e ‘cdr‘.

Para acessar os elementos de uma lista vamos usar o ‘first‘ e ‘rest‘.

Listas

- Vamos formalizar o conceito de lista
- Uma lista é
  - `empty`; ou
  - `(cons a lst)`, onde `a` é um valor qualquer e `lst` é uma lista
- Template baseado na definição

```
(define (fn-lista lst)
  (cond
    [(empty? lst) ...]
    [else ... (first lst)
          ... (fn-lista (rest lst)) ... ]))
```

Exemplo 1

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

Exemplo 2

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

Exemplo 3

Defina uma função que receba dois números naturais a e b como parâmetros e crie uma lista com todos os valores entre a e b.

Exemplo 4

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*.

Exemplo 5

Defina uma função que junte os elementos de duas listas de forma que os elementos fiquem na mesma ordem e os elementos da primeira lista fiquem antes dos elementos da segunda lista.

Listas aninhadas

Listas aninhadas

- As 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?

Listas aninhadas

- 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

- 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

- Template baseado na definição

```
(define (fn-lista-aninhada lst)
  (cond
    [(empty? lst) ...]
    [(list? (first lst))
     ... (fn-lista-aninhada (first lst))
     ... (fn-lista-aninhada (rest lst)) ...]
    [else ... (first lst)
          ... (fn-lista-aninhada (rest lst)) ... ]))
```

Exemplo 6

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

Exemplo 7

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

Árvores binárias

Árvores binárias

- Como podemos definir uma árvore binária?

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

- 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

- Template baseado na definição

```
(define (fn-arvore-bin t)
  (cond
    [(empty? t) ...]
    [else ... (arvore-bin-v t)
          ... (fn-arvore-bin (arvore-bin-dir t))
          ... (fn-arvore-bin (arvore-bin-esq t)) ...]))
```

Exemplo 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.