Fundamentos

Paradigma de Programação Funcional

Marco A L Barbosa

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

Conteúdo

- Expressões primitivas
- Combinações
- Avaliação de expressões
- Definições
- Modelo de substituição
- Condicional
- Operadores lógicos
- Recursão com números naturais

Expressões primitivas

Expressões primitivas

- Booleano
  - `#t` verdadeiro
  - `#f` falso
- Números
  - Exatos
    - Inteiros `1345`
    - Racionais `1/3`
    - Complexos com as partes real e imaginária exatas
  - Inexatos
    - Ponto flutuante `2.65`
    - Complexos com parte real ou imaginária inexata
- Muitos outros tipos de dados

Expressões primitivas

- Funções
  - Aritméticas: `+`, `-`, `*`, `/`
  - Relacionais: `>`, `>=`, `<`, `<=`, `=`
  - Muitas outras

Expressões primitivas

- A avaliação de um elemento (dado ou função) primitivo resulta no próprio elemento

```
> #t
#t
> 231
231
> +
#<procedure:+>
```

Combinações

Combinações

- Uma expressão que representa uma função pode ser combinada com expressões que representam dados para formar uma expressão que representa a aplicação da função aos dados

```
> (+ 12 56)
68
> (* 4 20)
90
> (> 4 5)
#f
```

- Este tipo de expressão é chamada de **combinação**

Combinações

- Uma combinação consiste de uma lista de expressões entre parênteses
  - A expressão mais a esquerda é chamada de **operador**
  - As outras expressões são os **operandos**
- O valor de uma combinação é obtido aplicando a função especificada pelo operador aos argumentos, que são os valores dos operandos

Combinações

- A convenção de colocar o operador a esquerda dos operandos é chamada de **notação pré-fixa**
- Vantagens da notação pré-fixa
  - Funções podem receber um número variado de argumentos

```
> (* 2 8 10 1)
160
```

  - Combinações podem ser aninhadas facilmente, isto é, os elementos das combinações podem também ser combinações

```
> (+ (* 3 5) (- 10 6) 5)
24
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

Avaliação de expressões

Avaliação de expressões

- Como o interpretador avalia uma expressão?

Avaliação de expressões

- Como o interpretador avalia uma expressão?

- Ele segue um procedimento
  - Se a expressão for um numeral, o valor é o número que o numeral representa
  - Se a expressão for uma função primitiva, o valor é a sequência de instrução de máquina para realizar a operação
  - Se a expressão for uma combinação
    - Avalie cada subexpressão da combinação
    - Aplique a função (valor da expressão mais a esquerda, operador) aos argumentos (valor das outras expressões, operandos)

Avaliação de expressões

- Observe que mesmo sendo um procedimento simples, ele pode ser usado para avaliar expressões muito complicadas (como por exemplo, expressões com muitos níveis de aninhamento)
- Isto é possível porque o procedimento é recursivo
- A recursão é uma ferramenta muito poderosa, e ela é essencial para a programação funcional

Avaliação de expressões

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
(+ (* 3 (+ 8 (+ 3 5))) (+ (- 10 7) 6))
(+ (* 3 (+ 8 8)) (+ (- 10 7) 6))
(+ (* 3 16) (+ (- 10 7) 6))
(+ 48 (+ (- 10 7) 6))
(+ 48 (+ 3 6))
(+ 48 9)
57
```

Definições

Definições

- Definições servem para dar nome a objetos computacionais, sejam dados ou funções
- É a forma de abstração mais elementar

Definições

- Em Racket, definições são feitas com o `define`

```
(define x 10)
(define y (+ x 24))
```

```
> y
34
```

- Quando o interpretador encontra uma construção do tipo `(define <nome> <exp>)`, ele associa `<nome>` ao valor obtido pela avaliação de `<exp>`
- A avaliação de um nome resulta no objeto associado a ele na sua definição
- A memória que armazena as associações entre nomes e objetos é chamada de **ambiente**

Definições

- O procedimento para avaliação de expressão não serve para definições
  - `(define x 10)` não significa aplicar a função `define` a dois argumentos, um o valor associado a `x` e o outro o valor `10`, o propósito do `define` é justamente associar o valor `10` a `x`
  - Ou seja, `(define x 10)` não é uma combinação
- Exceções a regra geral de avaliação de expressões são chamadas de **formas especiais**
- `define` é uma forma especial
- Cada forma especial tem a sua própria regra de avaliação
- O Racket possui poucas formas especiais, isto significa que é possível aprender a sintaxe da linguagem rapidamente

Definições

- Também é possível criar definições de novas funções
- A sintaxe geral é `(define (<nome> <parâmetros>) <corpo>)`
- Exemplos

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

(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))
```

```
> (quadrado 5)
25
> (quadrado (+ 2 6))
64
> (soma-quadrados (+ 2 2) 3)
25
```

- Observe que as funções compostas (definidas pelo usuário) são usadas da mesma forma que as funções pré-definidas

Modelo de substituição

Modelo de substituição

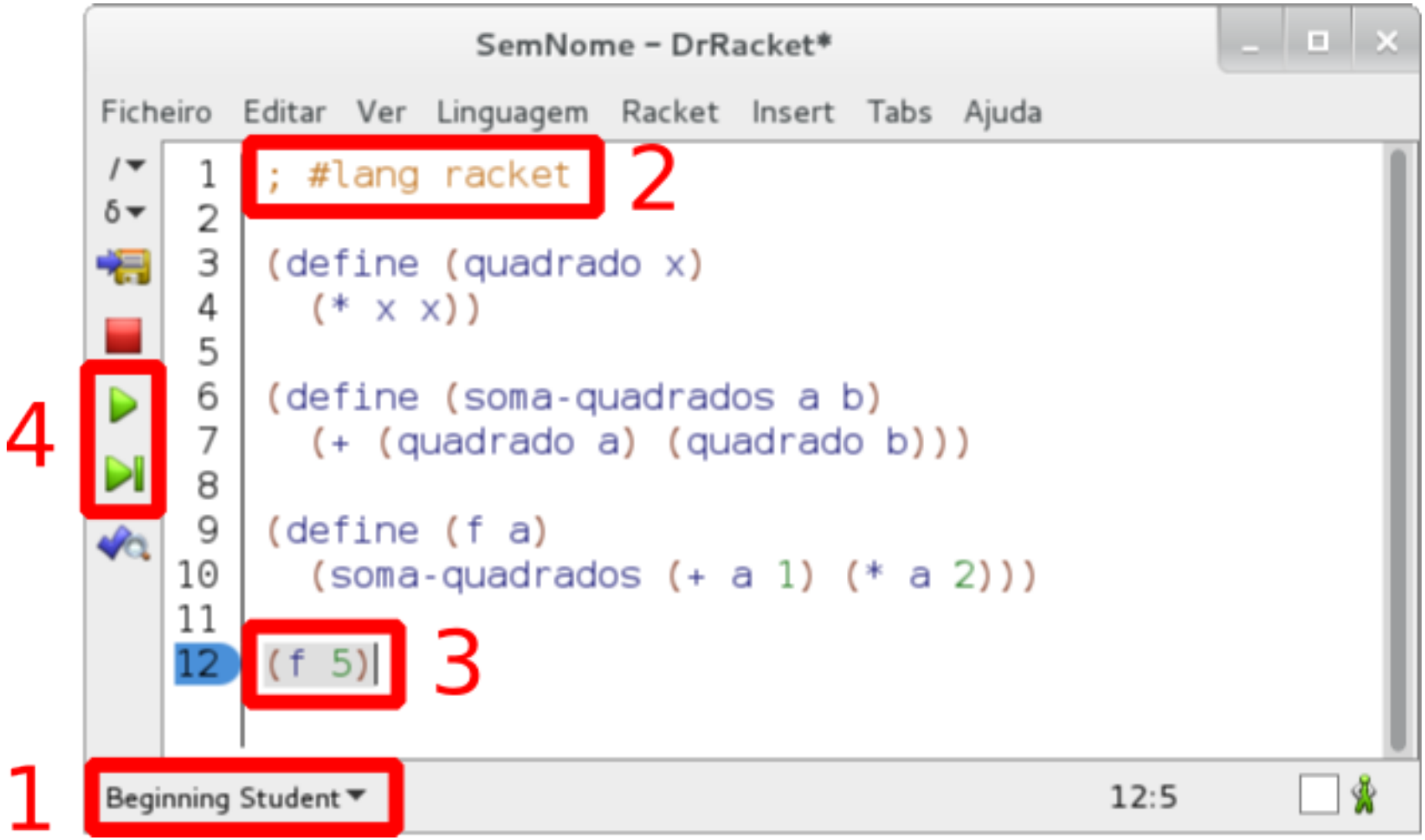
- A regra de avaliação de expressão que vimos anteriormente tem que ser estendida para contemplar o uso de funções compostas
- Para aplicar um função composta:
  - Avalie o corpo da função composta substituindo cada parâmetro formal pelo argumento correspondente
- Esta forma de aplicar funções compostas é chamada de **modelo de substituição**

Modelo de substituição

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

(define (soma-quadrados a b)
  (+ (quadrado a) (quadrado b)))

(define (f a)
  (soma-quadrados (+ a 1) (* a 2)))

(f 5)
(soma-quadrados (+ 5 1) (* 5 2))
(soma-quadrados 6 (* 5 2))
(soma-quadrados 6 10)
(+ (quadrado 6) (quadrado 10))
(+ (* 6 6) (quadrado 10))
(+ 36 (quadrado 10))
(+ 36 (* 10 10))
(+ 36 100)
136
```

Modelo de substituição

- Usando o DrRacket para fazer o passo a passo do modelo de substituição



Modelo de substituição

- Ao invés de avaliar os operandos e depois aplicar a função (fazer a substituição), existe um outro modo de avaliação que primeiro faz a substituição e apenas avalia os operandos quando (e se) eles forem necessários

```
(f 5)
(soma-quadrados (+ 5 1) (* 5 2))
(+ (quadrado (+ 5 1)) (quadrado (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* (* 5 2) (* 5 2)))
(+ 36 (* (* 5 2) (* 5 2)))
(+ 36 (* 10 (* 5 2)))
(+ 36 (* 10 10))
(+ 36 100)
136
```

- Observe que a resposta obtida foi a mesma do método anterior

Modelo de substituição

- Este método de avaliação alternativo de primeiro substituir e depois reduzir, é chamado de **avaliação em ordem normal** (que é um tipo de avaliação atrasada)
- O método de avaliação que primeiro avalia os argumentos e depois aplica a função é chamado de **avaliação em ordem aplicativa**
- O Racket usa por padrão a avaliação em ordem aplicativa
- O Haskell usa avaliação em ordem normal

Condicional

Condicional

- Vamos escrever uma função para calcular o valor absoluto de número, isto é

$$\mathrm{abs}(x) = \begin{cases} x & \text{se } x \geq 0 \\ \\ -x & \text{caso contrário} \end{cases}$$

- A forma especial `cond` é utilizada para especificar funções deste tipo

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [(< x 0) (- x)]))
```

```
(define (abs x)
  (cond
    [(>= x 0) x]
    [else (- x)]))
```

Condicional

- A forma geral do `cond` é

```
(cond
  (<p1> <e1>)
  (<p2> <e2>)
  (<p3> <e3>)
  ...
  [(else <en>)])
```

- Cada par (<p> <e>) é chamado de **cláusula**
- A primeira expressão de uma cláusula é chamada de **predicado**, isto é, uma expressão cujo o valor é interpretado como verdadeiro ou falso
- A segunda expressão de uma cláusula é chamada de **consequente**

Condicional

- Expressões condicionais são avaliadas da seguinte maneira
  - O predicado `p1` é avaliado, se o valor for falso, o predicado `p2` é avaliado, se o valor for falso, o predicado `p3` é avaliado, ...
  - Este processo continua até que um predicado cujo o valor seja verdadeiro é encontrado
  - Então o interpretador devolve o valor obtido pela avaliação do consequente associado com este predicado
  - `else` no `cond` é considerado um predicado verdadeiro
  - Se nenhum predicado verdadeiro for encontrado, um valor indefinido é devolvido

Condicional

- A forma especial `if` pode ser usada quando existem apenas dois casos
- A forma geral do `if` é
  - `(if <predicado> <consequente> <alternativa>)`
- Expressões `if` são avaliadas da seguinte maneira
  - O predicado é avaliado, se o resultado for verdadeiro, então o consequente é avaliado e o seu valor retornado
  - Caso contrário, a alternativa é avaliada e o seu valor retornado

Operadores lógicos

Operadores lógicos

- Predicados podem ser compostos usando as seguintes operações
  - `(and <e1> ... <en>)` - uma expressão por vez é avaliada da esquerda para a direita, se alguma `<e>` for falsa, o resultado do `and` é falso e o restante das expressões não são avaliadas. Se todas as expressões forem verdadeiras `and` é avaliado para o valor da última expressão
  - `(or <e1> ... <en>)` - uma expressão por vez é avaliada da esquerda para direita, se algum `<e>` for avaliada para um valor verdadeira, este valor é o resultado do `or` e o restante das expressões não são avaliadas. Se nenhuma expressão for verdadeira, `or` é avaliado para `#f`
  - `(not <e>)` - `#t` quando `<e>` for avaliado para um valor falso, e `#f` caso contrário
- Observe que `and` e `or` são formas especiais

Recursão com números naturais

Recursão com números naturais

Vamos usar a receita de projeto do livro HTDP (seção 9.4) como base para escrevermos funções recursivas

1. Contrato, propósito e cabeçalho
2. Exemplos
3. Template
4. Corpo
5. Teste

Recursão com números naturais

- Definição de número natural
  - 0 é um número natural
  - se n é um número natural, então $n + 1$ é um número natural
- Baseado nesta definição, criamos um template para funções com números naturais

```
(define (fn-natural n)
  (cond
    [(zero? n) ...]
    [else ... n (fn-natural (sub1 n))]))
```

Exemplo 1

Dado um número natural n, defina uma função `soma`, que some os números naturais até n.

Passo 1: Contrato, propósito e cabeçalho

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(define (soma n) 0)
```

Passo 1: Contrato, propósito e cabeçalho

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(define (soma n) 0)
```

Passo 2: Exemplos

```
(check-equal? (soma 0) 0)
(check-equal? (soma 1) 1) ; (+ 1 0)
(check-equal? (soma 3) 6) ; (+ 3 (+ 2 (+ 1 0)))
```

Passo 1: Contrato, propósito e cabeçalho

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(define (soma n) 0)
```

Passo 2: Exemplos

```
(check-equal? (soma 0) 0)
(check-equal? (soma 1) 1) ; (+ 1 0)
(check-equal? (soma 3) 6) ; (+ 3 (+ 2 (+ 1 0)))
```

Passo 3: Template (usamos o template para funções com números naturais)

```
(define (soma n)
  (cond
    [(zero? n) ...]
    [else ... n (soma (sub1 n))]))
```

Passo 4: Corpo (baseado nos exemplos, completamos o template)

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n

(check-equal (soma 0) 0)
(check-equal (soma 1) 1) ; (+ 1 0)
(check-equal (soma 3) 6) ; (+ 3 (+ 2 (+ 1 0)))

(define (soma n)
  (cond
    [(zero? n) ...]
    [else ... n (soma (sub1 n))]))
```

Passo 4: Corpo (baseado nos exemplos, completamos o template)

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n

(check-equal (soma 0) 0)
(check-equal (soma 1) 1) ; (+ 1 0)
(check-equal (soma 3) 6) ; (+ 3 (+ 2 (+ 1 0)))

(define (soma n)
  (cond
    [(zero? n) ...]
    [else ... n (soma (sub1 n))]))

(define (soma n)
  (cond
    [(zero? n) 0]
    [else ... n (soma (sub1 n))]))
```

Passo 4: Corpo (baseado nos exemplos, completamos o template)

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n

(check-equal (soma 0) 0)
(check-equal (soma 1) 1) ; (+ 1 0)
(check-equal (soma 3) 6) ; (+ 3 (+ 2 (+ 1 0)))

(define (soma n)
  (cond
    [(zero? n) ...]
    [else ... n (soma (sub1 n))]))

(define (soma n)
  (cond
    [(zero? n) 0]
    [else ... n (soma (sub1 n))]))

(define (soma n)
  (cond
    [(zero? n) 0]
    [else (+ n (soma (sub1 n)))]))
```

Código final

```
#lang racket

(require rackunit)
(require rackunit/text-ui)

;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(define soma-tests
  (test-suite
   "soma tests"
   (check-equal? (soma 0) 0)
   (check-equal? (soma 1) 1)  ; (+ 1 0)
   (check-equal? (soma 3) 6)); (+ 3 (+ 2 (+ 1 0)))

(define (soma n)
  (cond
    [(zero? n) 0]
    [else (+ n (soma (sub1 n)))]))
```

Código final - continuação

```
;; Lista de funções -> void
;; Executa um grupo de testes
(define (executar-testes . testes)
  (run-tests (test-suite "Todos os testes" testes))
  (void))

;; Chama a função para executar os testes
(executar-testes soma-tests)
```

Passo 5: testes (resultado após a execução `ctrl-r`)

```
3 success(es) 0 failure(s) 0 error(s) 3 test(s) run
```

Exemplo 2

Usando apenas as funções primitivas `zero?`, `add1` e `sub1`, defina a função `mais` que soma dois números naturais.

Passo 1: Contrato, propósito e cabeçalho

```
;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1
(define (mais a b) 0)
```

Passo 1: Contrato, propósito e cabeçalho

```
;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1
(define (mais a b) 0)
```

Passo 2: Exemplos

```
(check-equal? (mais 3 0) 3)
(check-equal? (mais 0 4) 4)
(check-equal? (mais 4 2) 6)
```

Passo 1: Contrato, propósito e cabeçalho

```
;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1
(define (mais a b) 0)
```

Passo 2: Exemplos

```
(check-equal? (mais 3 0) 3)
(check-equal? (mais 0 4) 4)
(check-equal? (mais 4 2) 6)
```

Passo 3: Template (usamos o template para funções com números naturais, vamos considerar que a recursão acontece no parâmetro b)

```
(define (mais a b)
  (cond
    [(zero? b) ...]
    [else ... b (mais a (sub1 b))]))
```

Passo 4: Corpo (baseado nos exemplos, completamos o template)

```
;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1

(check-equal? (mais 3 0) 3)
(check-equal? (mais 0 4) 4)
(check-equal? (mais 4 2) 6)

(define (mais a b)
  (cond
    [(zero? b) ...]
    [else ... b (mais a (sub1 b))]))
```

Passo 4: Corpo (baseado nos exemplos, completamos o template)

```
;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1

(check-equal? (mais 3 0) 3)
(check-equal? (mais 0 4) 4)
(check-equal? (mais 4 2) 6)

(define (mais a b)
  (cond
    [(zero? b) ...]
    [else ... b (mais a (sub1 b))]))

(define (mais a b)
  (cond
    [(zero? b) a]
    [else ... b (mais a (sub1 b))]))
```

Passo 4: Corpo (baseado nos exemplos, completamos o template)

```
;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1

(check-equal? (mais 3 0) 3)
(check-equal? (mais 0 4) 4)
(check-equal? (mais 4 2) 6)

(define (mais a b)
  (cond
    [(zero? b) ...]
    [else ... b (mais a (sub1 b))]))

(define (mais a b)
  (cond
    [(zero? b) a]
    [else ... b (mais a (sub1 b))]))

(define (mais a b)
  (cond
    [(zero? b) a]
    [else (add1 (mais a (sub1 b)))]))
```

Código final

```
#lang racket

(require rackunit)
(require rackunit/text-ui)

;; Natural Natural -> Natural
;; Soma dois números naturais usando as operações primitivas
;; zero?, add1 e sub1
(define mais-tests
  (test-suite
   "soma tests"
   (check-equal? (mais 3 0) 3)
   (check-equal? (mais 0 4) 4)
   (check-equal? (mais 4 2) 6)))

(define (mais a b)
  (cond
    [(zero? b) a]
    [else (add1 (mais a (sub1 b)))]))
```

Código final - continuação

```
;; Executa um grupo de testes
(define (executar-testes . testes)
  (run-tests (test-suite "Todos os testes" testes))
  (void))
;; chama a função para executar os testes
(executar-testes mais-tests)
```

Passo 5: testes (resultado após a execução `ctrl-r`)

```
3 success(es) 0 failure(s) 0 error(s) 3 test(s) run
```

Exemplo 3

Usando apenas as funções primitivas `zero?`, `add1` e `sub1`, defina a função `menor` que verifica se um número natural é menor que outro.

Exemplo 4

[htdp 11.4.7] Escreva uma função `nao-divisivel-<=i`, que receba como parâmetros dois números naturais, i e n, onde $i < n$. Se n não é divisível por nenhum número entre 1 (exclusive) e i (inclusive), a função deve devolver verdadeiro, caso contrário falso. Utilizando a função `nao-divisivel-<=i`, defina uma função `primo?`, que verifica se um número natural é primo. Um número natural é primo se ele tem exatamente dois divisores distintos: 1 e ele mesmo.