Fundamentos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

O paradigma de programação funcional é baseado na definição e aplicação de funções

- As funções são escritas em termos de expressões
- Todas expressões produzem um resultado quando são avaliadas

Mas o que são expressões e como o interpretador as avalia?

Definição de expressão (versão 0.1)

Uma expressão consiste de

- Um literal; ou
- Uma função primitiva

Literais de tipos primitivos

Números Exatos

- Inteiros `1345`
- Racionais `1/3`
- Complexos com as partes real e imaginária exatas

Números Inexatos

- Ponto flutuante `2.65`
- Complexos com parte real ou imaginária inexata

Literais de tipos primitivos

Booleano

- `#t` verdadeiro
- `#f` falso

Strings

- `"Seu nome"`

Muitos outros tipos

Funções primitivas

Aritméticas: `+, -, *, /`

Relacionais: `>, >=, <, <=, =`

Strings: **`string-length`**, **`string-append`**, **`number->string`**, **`string->number`**

Muitas outras...

Processo de avaliação de expressões (versão 0.1)

Uma expressão consiste de

- Um literal; ou
- Uma função primitiva

Como uma expressão é avaliada?

- Literal $\rightarrow$ valor que o literal representa
- Função primitiva $\rightarrow$ sequência de instruções de máquina associada com a função

Como a regra de avaliação de uma expressão está ligada com a definição de expressão?

Uma expressão é definida em termos de dois casos e por isso a regra de avaliação de expressões também é definida por dois casos.

Exemplo de avaliação de expressões

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

A definição de expressão que acabamos de ver parece bastante limitada, o que está faltando?

Uma forma de combinar expressões para formar novas expressões!

Combinações

Uma expressão que é avaliada para uma função pode ser combinada com outras expressões para formar uma expressão que representa a aplicação da função a estas expressões. Ah? Ok, vamos ver alguns exemplos

```
> (+ 12 56)
68
> (* 4 20)
80
> (> 4 5)
#f
> (string-append "Apenas " "um " "teste")
"Apenas um teste"
```

Uma **combinação** consiste de uma lista de expressões entre parênteses

- A expressão mais a esquerda é o **operador**
- As outras expressões são os **operandos**

Qual o valor de uma combinação?

O resultado da aplicação da função especificada pelo operador aos valores dos operandos (argumentos).

Que tipo de notação é essa, parece estranha!

A convenção de colocar o operador a esquerda dos operandos é chamada de **notação prefixa**.

Quais as vantagens e desvantagens da notação prefixa?

Operadores aritméticos são tratados como as outras funções é podem receber um número variado de argumentos

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

Vantagens da notação prefixa

Combinações podem ser aninhadas facilmente, sem preocupações com prioridades das operações

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

Desvantagens da notação prefixa

Diferente da forma que aprendemos...

Pode requerer muitos parênteses.

Vamos atualizar a definição de expressão para incluir as combinações.

Definição de expressão (versão 0.2)

Uma expressão consiste de

- Um literal; ou
- Uma função primitiva; ou
- Uma combinação (lista de **expressões** entre parênteses)

Como uma expressão é avaliada?

- Literal $\longrightarrow$ valor que o literal representa
- Função primitiva $\longrightarrow$ sequência de instruções de máquina associada com a função
- Combinação
  - **Avalie cada expressão** da combinação, isto é, reduza cada expressão para um valor

    $\longrightarrow$ Resultado da aplicação da função aos argumentos

Algumas observações interessantes

- Uma expressão é definida por três casos e a regra de avaliação também tem três casos.
- Quando uma expressão é uma combinação, ela contém outras expressões. Quando uma definição refere-se a si mesmo, dizemos que ela é uma definição com **autorreferência**. O uso de autorreferência permite que expressões de tamanhos arbitrários sejam criadas.
- O processo de avaliação para uma expressão que é uma combinação requer a chamada do processo de avaliação para suas expressões. Quando um processo é definido em termos de si mesmo, dizemos que ele é **recursivo**. O uso de recursividade permite que expressões de tamanho arbitrário sejam avaliadas.
- Uma autorreferência em uma definição implicada (geralmente) em uma recursão para processar os elementos que seguem a definição.

Estamos usando os conceitos de autorreferência e recursividade para entender o funcionamento da linguagem Racket (a estrutura das linguagens de programação são recursivas), mas iremos ver que estes conceitos são fundamentais também para criar programas no paradigma funcional.

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

Vimos anteriormente que o paradigma de programação funcional é baseado na definição e aplicações de funções.

Como funções são definidas em termos de expressões, nós vimos primeiramente o que são expressões.

Agora vamos ver o que são definições e como fazer definições de novas funções.

Definições

Qual o propósito das definições?

Definições servem para dar nome a objetos computacionais, sejam dados ou funções.

- É a forma de abstração mais elementar

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

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

```
> y
34
```

Como o Racket interpreta um definição?

Quando o interpretador encontra uma construção do tipo

```
(define <nome> <exp>)
```

ele associa `<nome>` ao valor obtido pela avaliação de `<exp>` (a memória que armazena as associações entre nomes e objetos é chamada de **ambiente**).

Note que uma definição não é uma combinação (expressão) e por isso o procedimento para avaliação de expressão não serve para definições.

- `(define x 10)` não significa aplicar a função `define` a dois argumentos
- O propósito do `define` é associar o valor `10` ao nome `x`
- Ou seja, `(define x 10)` não é uma combinação (expressão)

Dessa forma, os programas em Racket são compostos de duas construções: expressões e definições.

De forma mais precisa, um programa em Racket é formado por uma sequência de definições e expressões.

Como vimos na definição `(define y (+ x 24))`, nomes podem aparecer em expressões, então precisamos atualizar a nossa definição de expressão. Mas antes, vamos ver como definir novas funções.

A sintaxe geral para definição de novas funções (**funções compostas**) é

```
(define (<nome> <parametro>...) <exp>)
```

```
(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.

Agora precisamos estender a definição de expressões para incluir nomes e alterar a regra de avaliação de expressões para considerar a aplicação de funções compostas.

Modelo de substituição

Definição de expressão (versão 0.3)

Uma expressão consiste de

- Um literal; ou
- Uma função primitiva; ou
- Um nome; ou
- Uma combinação

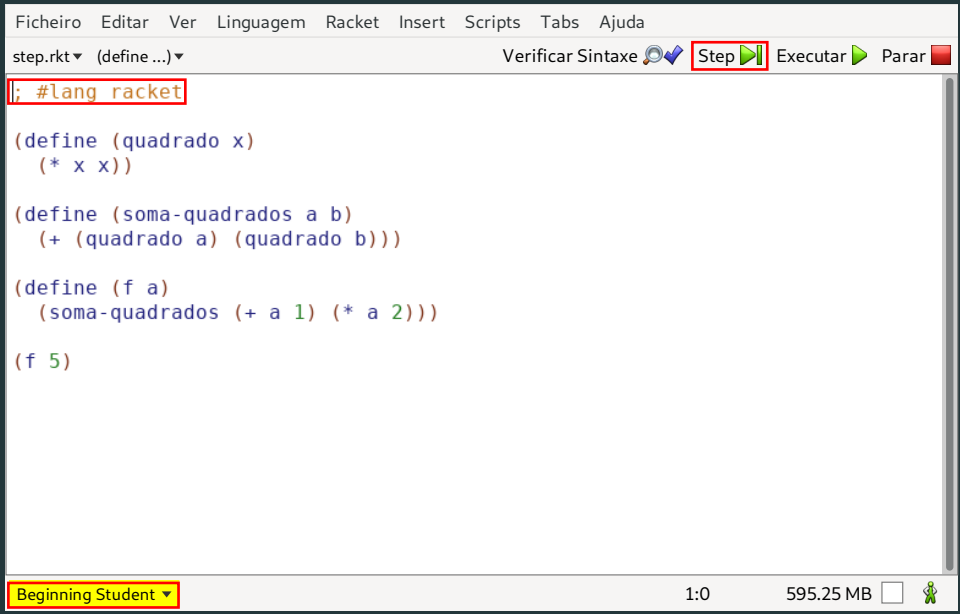
Avaliação

- Literal $\rightarrow$ valor que o literal representa
- Função primitiva $\rightarrow$ sequência de instruções de máquina associada com a função
- Nome $\rightarrow$ valor associado com o nome no ambiente
- Combinação
  - **Avalie** cada expressão da combinação
  - Se o operador é uma função composta , **avalie** o corpo da função composta **substituindo** cada ocorrência do parâmetro formal pelo argumento correspondente
  - Senão, aplique a função primitiva aos argumentos

Essa forma de calcular o resultado da aplicação de 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)                           ; Substitui (f 5) pelo corpo de f com
                                ; as ocorrências do parâmetro a
                                ; substituídas pelo argumento 5

(soma-quadrados (+ 5 1) (* 5 2)); Reduz (+ 5 1) para o valor 6

(soma-quadrados 6 (* 5 2))      ; Reduz (* 5 2) para o valor 10

(soma-quadrados 6 10)           ; Subs (soma-quadrados 6 10) pelo corpo ...

(+ (quadrado 6) (quadrado 10))  ; Subs (quadrado 6) pelo corpo ...

(+ (* 6 6) (quadrado 10))       ; Reduz (* 6 6) para 36

(+ 36 (quadrado 10))            ; Subs (quadrado 10) pelo corpo ...

(+ 36 (* 10 10))                ; Reduz (* 10 10) para 100

(+ 36 100)                      ; Reduz (+ 36 100) para 136

136
```


Ficheiro Editar Ver Linguagem Racket Insert Scripts Tabs Ajuda
step.rkt (define ...)
Verificar Sintaxe
Step
Executar
Parar
; #lang racket
(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)
Beginning Student
1:0
595.25 MB

Ao invés de avaliar os operandos e depois 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.

Este método de avaliação alternativo de primeiro substituir e depois reduzir, é chamado de **avaliação em ordem normal** (que é um tipo de avaliação preguiçosa).

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.

1. O seu amigo Alan está planejando uma viagem pro final do ano com a família e está considerando diversos destinos. Uma das coisas que ele está levando em consideração é o custo da viagem, que inclui, entre outras coisas, hospedagem, combustível e o pedágio. Para o cálculo do combustível ele pediu a sua ajuda, ele disse que sabe a distância que vai percorrer, o preço do litro do combustível e o rendimento do carro (quantos quilômetros o carro anda com um litro de combustível), mas que é muito chato ficar fazer o cálculo manualmente, então ele quer que você faça um programa para calcular o gasto de combustível em uma viagem.

```
(define (custo-combustivel distancia preco-do-litro rendimento)
    (* (/ distancia rendimento) preco-do-litro))
```

2. Depois que você fez o programa para o Alan, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

É possível resolver este problema usando as coisas que vimos até aqui? Não!

O que está faltando? Algum tipo de expressão condicional.

Depois voltamos nesse problema!

Condicional

Utilizamos a construção **if** para especificar expressões condicionais. Sua forma geral é

```
(if <predicado> <consequente> <alternativa>)
```

Exemplos

```
> (if (> 4 2) 10 7)
10
> (if (= 10 12) 10 7)
7
```

Qual a diferente do **`if`** do Racket em relação ao das outras linguagens?

O **`if`** do Racket é uma expressão, ele produz um valor como resultado. Na maioria das outras linguagens o **`if`** é uma sentença, ele não produz um resultado mas gera uma mudança no estado do programa.

O **`if`** é uma função? Não.

Se o **`if`** fosse uma função ele seria avaliado usando a regra de avaliação de funções, que diz que todas as expressões dos argumentos da função devem ser avaliados antes da aplicação na função. O **`if`** avalia o consequente ou a alternativa, dependendo da condição, mas não os dois.

O **`if`** é uma **forma especial** e tem uma regra de avaliação específica. (O Racket possui poucas formas especiais, isto significa que é possível aprender a sintaxe da linguagem rapidamente.)

Regra de avaliação do `if`

Expressões **`if`** são avaliadas da seguinte maneira:

- Se o predicado não é um valor, avalie o predicado e o substitua pelo seu valor
- Se o predicado é `#t`, substitua toda a expressão **`if`** pelo consequente e avalie o consequente
- Se o predicado é `#f`, substitua toda a expressão **`if`** pela alternativa e avalie a alternativa

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

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

e ver o processo de avaliação dessa função.

```
(define (abs x)
  (if (>= x 0)
      x
      (- x)))

(abs -4)          ; Substitui (abs -4) pelo corpo ...

(if (>= -4 0)     ; Como o predicado não é um valor,
    -4            ; a expressão (>= -4 0) é avaliada e
    (- -4))       ; substituída pelo seu valor

(if #f            ; Como o predicado é #f, a expressão if
    -4            ; é substituída pela alternativa
    (- -4))       ;

(- -4)            ; Reduz (- -4) para 4 - não mostrado...
```

Definição de expressão (versão 1.0)

Vamos atualizar a nossa definição de expressão pra incluir formas especiais.

Definição de expressão (versão 1.0)

Uma expressão consiste de

- Um literal; ou
- Uma função primitiva; ou
- Um nome; ou
- Uma forma especial; ou
- Uma combinação

Avaliação

- Literal $\longrightarrow$ valor que o literal representa
- Função primitiva $\longrightarrow$ sequência de instruções de máquina associada com a função
- Nome $\longrightarrow$ valor associado com o nome no ambiente
- Forma especial $\longrightarrow$ avalie a forma especial usando a regra de avaliação específica
- Combinação
  - **Avalie** cada expressão da combinação
  - Se o operador é uma função composta, **avalie** o corpo da função composta **substituindo** cada ocorrência do parâmetro formal pelo argumento correspondente
  - Senão, aplique a função primitiva aos argumentos

A forma especial `cond` pode ser usada quando existem vários (pelo menos um) casos

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

Como as duas condições são mutuamente excludentes, podemos usar o `else`

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

A forma geral do `cond` é

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

Cada par `[<p> <e>]` é chamado de **cláusula** (parênteses e colchetes são equivalentes em Racket).

A primeira expressão de uma cláusula é chamada de **predicado** (expressão cujo o valor é interpretado como verdadeiro ou falso).

A segunda expressão de uma cláusula é chamada de **consequente**.

Expressões **`cond`** são avaliadas da seguinte maneira:

- Se o primeiro predicado não é um valor, avalie o predicado e o substitua pelo seu valor. Ou seja, substitua todo o **`cond`** por um novo **`cond`** onde o primeiro predicado foi substituído pelo seu valor
- Se o primeiro predicado é `#t` ou **`else`**, substitua a expressão `cond` inteira pelo primeiro consequente e avalie o consequente
- Se o primeiro predicado é `#f`, remova a primeira cláusula. Isto é, substitua o **`cond`** por um novo **`cond`** sem a primeira cláusula
- Se não tem mais cláusula, sinalize um erro

Condicional

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

(abs -4)          ; Substitui (abs -4) pelo corpo ...

(cond             ; Como o primeiro predicado não é um valor,
  [(>= -4 0) -4] ; a expressão (>= -4 0) é avaliada
  [else (- -4)]) ; e substituída pelo seu valor

(cond             ; Como o primeiro predicado é falso, a primeira
  [#f -4]         ; cláusula é removida
  [else (- -4)]) ;

(cond             ; Como o primeiro predicado é else,
  [else (- -4)]) ; o cond é substituído pelo primeiro consequente

(- -4)            ; Reduz (- -4) para 4

4
```

Exercício

Defina as funções `e-logico` e `ou-logico` de tal forma que para os argumentos x e y:

`(e-logico x y)` $\rightarrow x \wedge y$

`(ou-logico x y)` $\rightarrow x \vee y$

```
(define (e-logico x y)
  (if x
    y
    #f))

(define (ou-logico x y)
  (if x
    #t
    y))
```

Operadores lógicos

Operadores lógicos

Predicados podem ser compostos usando as formas especiais **`and`** e **`or`** e a função **`not`**

not

A função `(not <e>)` produz `#t` quando `<e>` for avaliado para um valor falso, e `#f` caso contrário

```
> (not (> 5 2))
#f
> (not (< 5 2))
#t
```

and

A forma geral do **and** é:

```
(and <e1> ... <en>)
```

and

Expressões **and** são avaliadas da seguinte maneira

- Se não existem expressões, produza `#t`
- Se a primeira expressão não é um valor, avalie a primeira expressão e a substitua pelo seu valor
- Se a primeira expressão é `#f`, produza `#f`
- Se a primeira expressão é `#t`, substitua a expressão **and** por uma nova expressão `and` sem a primeira expressão
- **Observação**: o passo a passo do Racket é um pouco diferente (não elimina os valores `#t`)

and

```
(and (> 4 2) #t (= 3 3))   ; A primeira expressão não é um valor,
                           ; logo ela é avalida e substiuída pelo
                           ; seu valor

(and #t #t (= 3 3))        ; A primeira expressão é #t, então
                           ; ela é removida do and

(and #t (= 3 3))           ; A primeira expressão é #t, então
                           ; ela é removida do and

(and (= 3 3))              ; Reduz (= 3 3) para #t

(and #t)                   ; A primeira expressão é #t, então
                           ; ela é removida do and

(and )                     ; Não tem mais expressões, produz #t

#t
```

or

A forma geral do **or** é:

```
(or <e1> ... <en>)
```

or

Expressões **or** são avaliadas da seguinte maneira

- Se não existem expressões, produza `#f`
- Se a primeira expressão não é um valor, avalie a primeira expressão e a substitua pelo seu valor
- Se a primeira expressão é `#t`, produza `#t`
- Se a primeira expressão é `#f`, substitua a expressão **or** por uma nova expressão **or** sem a primeira expressão

Observação: o passo a passo do Racket é um pouco diferente (não elimina os valores `#f`)

Operadores lógicos

```
(or (< 4 2) #t (= 3 3))  ; A primeira expressão não é um valor,
                         ; logo ela é avalida e substituída pelo
                         ; seu valor

(or #f #t (= 3 3))       ; A primeira expressão é #f, então
                         ; ela é removida do or

(or #t (= 3 3))          ; A primeira expressão é #t; produz #t

#t
```

Operadores de equivalência

Operadores de equivalência

Os operadores de equivalência são utilizados para verificar a relação de equivalência entre expressões.

- Não devem ser confundidos com o comparador =, utilizado apenas para valores numéricos
- Os principais operadores de equivalência são o **`eq?`**, **`eqv?`** e **`equal?`**

Operador eq?

A função `(eq? v1 v2)` produz `#t` se `v1` e `v2` referenciam o mesmo objeto, `#f` caso contrário. **`eq?`** é avaliada rapidamente pois compara apenas as referências. Entretanto, o **`eq?`** pode não ser adequado, pois a geração dos objetos pode não ser clara

```
> (eq? 2 2)
#t
> (eq? (+ 3 5) (+ 5 3))
#t
> (eq? 2 2.0)
#f
> (eq? (expt 2 100) (expt 2 100))
#f
> (eq? (integer->char 955) (integer->char 955))
#f
```

Observe que nos três últimos exemplos, objetos distintos foram criados para expressões avaliadas para um mesmo valor.

Operador eqv?

Dois valores são **eqv?** *sse* eles são **eq?**, exceto para números e caracteres.

Dois números são **eqv?** se eles são precisamente iguais

```
> (eqv? (expt 2 100) (expt 2 100))
#t
> (eqv? 2 2.0)
#f
```

Operador eqv?

Dois caracteres são **eqv?** quando seus resultados de **char->integer** forem iguais

```
> (eqv? (integer->char 955) (integer->char 955))
#t
> (eqv? #\a #\z)
#f
```

Operador eqv?

Dois pares iguais não são **eqv?** entre si (recai ao **eq?**)

```
> (eqv? (cons 1 2) (cons 1 2))
#f
```

Operador `equal?`

Dois valores são **`equal?`** *sse* eles são **`eqv?`**, a menos que especificado de outra forma para um tipo de dado particular.

Duas strings são **`equal?`** quando elas possuem o mesmo tamanho e contêm a mesma sequência de caracteres

```
> (equal? "banana" "banana")
#t
> (equal? "banana" "abacaxi")
#f
```

Operador `equal?`

Para estruturas que podem ser compostas, como pares, vetores e etc, o operador **`equal?`** checa a equivalência recursivamente

```
> (equal? (list 3 (list 4 2) 5) (list 3 (list 4 2) 5))
#t
> (equal? (list 3 2.0 1) (list 3 2 1))
#f
```

Referências

Referências

Básicas

- Introdução rápida ao Racket
- Capítulos 1 e 2 (2.1 e 2.2) do Guia Racket
- Seção 1.1 do livro SICP

Complementares

- Capítulos 1 e 2 do livro TSLP4