Funcional II: recursão e iteração

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

26 de junho de 2018

Crédito de parte das imagens, a menos se especificado: Wikipedia

Objetivos

- Conhecer técnicas de modularização e teste de unidade em Racket
- Praticar a programação de processos recursivos
- Aprender a otimizar problemas sobrepostos com memoização
- Praticar a programação de processos iterativos com recursão na cauda

Modularização básica em Racket

Modularização em Racket

- Duas técnicas principais
  1. **Módulos locais**: restritos a um projeto ou a um conjunto de projetos
  2. **Módulos globais**: disponíveis para o ambiente (i.e. instaláveis via `raco pkg install`)
- Vamos analisar apenas a técnica de módulos locais
- A técnica de módulos globais fica como exercício

Módulos locais em Racket

Convenções e sintaxe

1. **Convenção**[1]: todo módulo reside em seu próprio arquivo (i.e. `.rkt`)
2. **Cabeçalho**: todo módulo inicia com a diretiva `#lang` (no nosso caso, especificando Racket):

```
1 | #lang racket
```

3. **Dependências providas**: exportamos definições via a forma `provide`:

```
1 | (provide factorial)
```

4. **Dependências requeridas**: importamos o módulo via a forma `require` parametrizada com o nome do arquivo (todas as dependências providas pelo arquivo são importadas):

```
1 | (require "factorial.rkt")
```

Nota: para módulos locais, o nome da dependência requerida é resolvido via caminho relativo (estilo Unix) ao diretório atual. Exemplos:

```
(require "libs/factorial.rkt")
(require "../factorial.rkt")
```

[1] Convenção pois é possível incluir mais de um módulo por arquivo, por exemplo usando a forma `module`

Testes de unidades com RackUnit

RackUnit: testes de unidades

Revisão e definições

- Duas principais técnicas de teste:
  - Testes baseados em propriedades (i.e. QuickCheck)
  - Testes baseados em exemplos (i.e. testes de unidade[2] ou *unit tests*)
- Testes de unidades podem ser efetuados:
  - De modo *ad-hoc* (i.e. com infra-estrutura própria do desenvolvedor)
  - Com o auxílio de frameworks de teste
- Vantagens dos frameworks de teste:
  - Simplificação
  - Automação
  - Documentação (*reporting*)
- **RackUnit**: framework de testes de unidade em Racket
  - Vamos aprender a usar tendo a função fatorial como exemplo...

[2] Unidade, nessa definição, se refere a um módulo coeso de software, que pode ser uma função, uma classe, um pacote, etc.

Exemplo 1: módulo fatorial

Definição e teste

- Para praticar modularização, vamos definir um módulo fatorial (`factorial.rkt`)[3]:

```
1  #lang racket
2
3  (provide factorial)
4
5  (define (factorial n)
6    (if (< n 0)
7        (raise-argument-error 'factorial "positive integer" n)
8        (factorial-iter n n 1)))
9
10 (define (factorial-iter n i fat)
11   (if (= i n)
12       fat
13       (factorial-iter n (sub1 i) (* fat i))))
```

- Definir as verificações de teste (**num arquivo separado:** `factorial-tests.rkt`):

```
1  #lang racket
2
3  (require rackunit "factorial.rkt")
4
5  (check-equal? (factorial 0)  1 "0! = 1")
6  (check-equal? (factorial 1)  1 "0! = 1")
7  (check-equal? (factorial 2)  2 "2! = 2")
8  (check-equal? (factorial 3)  6 "3! = 6")
9  (check-equal? (factorial 10) 3628800 "10! = 3628800")
10 (check-equal? (factorial 22) 1124000727777607680000 "22! = 1124000727777607680000")
```

[3] Para mais detalhes sobre disparo de exceções em Racket, consulte o guia:
`https://docs.racket-lang.org/guide/exns.html`

Módulo fatorial: resultado dos testes

Exemplo 1

```
$ racket factorial-test.rkt
--------------------
FAILURE
name:       check-equal?
location:   factorial-test.rkt:10:0
message:    "2! = 2"
actual:     1
expected:   2
--------------------
--------------------
FAILURE
name:       check-equal?
location:   factorial-test.rkt:11:0
message:    "3! = 6"
actual:     1
expected:   6
--------------------
--------------------
FAILURE
name:       check-equal?
location:   factorial-test.rkt:12:0
message:    "10! = 3628800"
actual:     1
expected:   3628800
--------------------
--------------------
FAILURE
name:       check-equal?
location:   factorial-test.rkt:13:0
message:    "22! = 1124000727777607680000"
actual:     1
expected:   1124000727777607
```

Módulo fatorial: correção e teste

Exemplo 1

- Correção da função fatorial:

```
1 (define (factorial-iter n i fat)
2   (if (= i 0)
3       fat
4       (factorial-iter n (sub1 i) (* fat i))))
```

- Re-execução dos testes:

```
$ racket factorial-test.rkt
```

Se nada é impresso, todos os testes passaram

Módulo fatorial: casos de teste

Exemplo 1

- Podemos agrupar múltiplas verificações (preferencialmente correlacionadas) em um *caso de teste*, que pode possuir um *título*
- Vantagem: se uma verificação do caso falha, as próximas não serão executadas

```
1  #lang racket
2
3  (require rackunit "factorial.rkt")
4
5  (test-case
6    "Test individual factorial values"
7    (let ([n '(0 1 2 3 10 22)]
8          [nf '(1 1 2 6 3628800 1124000727777607680000)])
9      (map check-equal? (map factorial n) nf)
10     (void)))
```

- Supondo que `factorial-iter` ainda tem o erro inicial:

```
$ racket factorial-test.rkt
--------------------
Test individual factorial values
FAILURE
name:       check-equal?
location:   factorial-test.rkt:20:9
actual:     1
expected:   2
--------------------
```

Módulo fatorial: integração com QuickCheck

Exemplo 1

- Na aula passada, vimos uma possibilidade de testar fatorial com base em propriedades
- Podemos integrar testes do QuickCheck no RackUnit:

```
1  #lang racket
2
3  (require rackunit "factorial.rkt")
4  (require quickcheck)
5  (require rackunit/quickcheck)
6
7  (...)
8
9  (define fatorial-n-fatorial-n+1
10   (property ([n arbitrary-natural])
11             (= (* (add1 n) (factorial n)) (factorial (add1 n)))))
12
13 (check-property fatorial-n-fatorial-n+1)
```

```
$ racket factorial-test.rkt
(...)
--------------------
FAILURE
name:       check-property
location:   factorial-test.rkt:30:0
params:     '(#<property>)
ntest:      0
stamps:     ()
arguments:  "n = 2"

Falsifiable
--------------------
```

Módulo fatorial: definição de suites

Exemplo 1

- À medida que nosso conjunto de testes aumenta, é interessante organizá-los em suítes
- Suítes podem conter testes individuais, casos ou outras suítes
  - Se um teste falhar, os seguintes não executam
- Executa todos os componentes (a menos que encontre um erro) e provê um relatório no final
- Porém, suítes não executam sozinhas, precisamos ativá-las via uma UI (no caso, vamos usar a `text-ui`)

Módulo fatorial: definição de suites

Exemplo 1

```
1  (require rackunit "factorial.rkt")
2  (require quickcheck)
3  (require rackunit/quickcheck)
4
5  (require rackunit/text-ui)
6
7  (define factorial-suite
8    (test-suite
9      "Equality and property tests for factorial"
10     ; test case
11     (test-case
12       "Test individual factorial values"
13       (let ([n '(0 1 2 3 10 22)]
14             [nf '(1 1 2 6 3628800 1124000727777607680000)])
15         (map check-equal? (map factorial n) nf)
16         (void)))
17
18     ; property-based test
19     (let ([fatorial-n-fatorial-n+1
20            (property ([n arbitrary-natural])
21                  (= (* (add1 n) (factorial n)) (factorial (add1 n))))])
22       (check-property fatorial-n-fatorial-n+1)
23       (void))))
24
25 (run-tests factorial-suite)
```

Módulo fatorial: definição de suites

Exemplo 1

- Com erro:

```
$ racket factorial-test.rkt
--------------------
Equality and property tests for factorial > Test individual factorial values
FAILURE
name:       check-equal?
location:   factorial-test.rkt:27:13
actual:     1
expected:   2
--------------------

--------------------
Equality and property tests for factorial > Unnamed test
FAILURE
name:       check-property
location:   factorial-test.rkt:34:6
params:     '(#<property>)
ntest:      0
stamps:     ()
arguments:  "n = 2"

Falsifiable
--------------------
0 success(es) 2 failure(s) 0 error(s) 2 test(s) run
2
```

- Após corrigirmos o erro:

```
$ racket factorial-test.rkt
OK, passed 100 tests.
2 success(es) 0 failure(s) 0 error(s) 2 test(s) run
0
```

Exercício 1

Construa um programa que aproxime o número de Euler por meio da série convergente de recíprocos da função fatorial:

$$\sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \cdots = e$$

O erro de aproximação deve ser um parâmetro da função. Por exemplo, para um erro de 4 casas decimais, chamamos:

```
(euler 0.0001)
```

Exercício1

Solução 1

```
1  #lang racket
2
3  ; approximate the euler number via infinite series 1/n!
4
5  (require "factorial.rkt")
6
7  (provide euler)
8
9  (define (euler err)
10   (let euler-iter ([n 0] [p-e 1.0] [c-e 0.0])
11     (if (< (abs (- p-e c-e)) err)
12         c-e
13         (euler-iter (add1 n) c-e (+ c-e (/ 1.0 (factorial n)))))))
```

- Desvantagem: a solução envolve muitos recálculos devido a *subproblemas sobrepostos*:

$$e \approx \frac{1}{1} + \frac{1}{1 \cdot 1} + \frac{1}{1 \cdot 1 \cdot 2} + \frac{1}{1 \cdot 1 \cdot 2 \cdot 3} + \frac{1}{1 \cdot 1 \cdot 2 \cdot 3 \cdot 4} + \frac{1}{1 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5} + \cdots$$

Exercício1

Solução 2

- Para evitar o recálculo dos problemas sobrepostos, podemos *memoizar* os valores de fatorial
- Como a sequência é acumulativa dependente apenas do último valor, basta memoizarmos o último valor e atualizar a cada iteração

```
1 (define (fast-euler err)
2   (let euler-iter ([n 0] [p-e 1.0] [c-e 0.0] [factorial 1])
3     (if (< (abs (- p-e c-e)) err)
4         c-e
5         (euler-iter (add1 n)
6                     c-e
7                     (+ c-e (/ 1.0 factorial))
8                     (* factorial (add1 n))))))
```

Exercício 2

A sequência de Fibonacci é definida pela seguinte função:

$$\mathrm{Fib}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & n > 2 \end{cases}$$

a) Construa um procedimento recursivo que determine o n-ésimo número de Fibonacci.

b) Proponha uma definição iterativa da sequência e construa o procedimento que a implemente. O procedimento deverá apresentar recursão na cauda.

c) Monitore via `trace` os algoritmos dos itens anteriores e verifique qual gera menos overhead espacial na pilha de execução. Adicionalmente, monitore o tempo de execução das duas versões via função `time`.

Exercício 2

Parte 1: solução 1

- Se modelarmos um processo recursivo, o procedimento resultante é trivial:

```
1 (define (fibonacci-recur n)
2   (cond [(= n 1) 1]
3         [(= n 2) 1]
4         [else (+ (fibonacci-recur (- n 1))
5                  (fibonacci-recur (- n 2)))]))
```

- A desvantagem desse algoritmo é a ineficiência:

```
> (time (fibonacci-recur 40))
cpu time: 1122 real time: 1122 gc time: 0
102334155
```

- A ineficiência[4] se deve à presença de recursão em árvore
- Implica a solução de subproblemas sobrepostos

[4] A complexidade da versão recursiva é $O((1.6180)^n)$ em tempo e $O(n)$ em espaço

Exercício 2

Parte 1: solução 1

```
> (fibonacci-recur 6)
>(fibonacci-recur 6)
> (fibonacci-recur 5)
> >(fibonacci-recur 4)
> > (fibonacci-recur 3)
> > >(fibonacci-recur 2)
< < <1
> > >(fibonacci-recur 1)
< < <1
< < 2
> > (fibonacci-recur 2)
< < 1
< <3
> >(fibonacci-recur 3)
> > (fibonacci-recur 2)
< < 1
> > (fibonacci-recur 1)
```

```
< < 1
< <2
< 5
> (fibonacci-recur 4)
> >(fibonacci-recur 3)
> > (fibonacci-recur 2)
< < 1
> > (fibonacci-recur 1)
< < 1
< <2
> >(fibonacci-recur 2)
< <1
< 3
<8
8
```

- Note que:
  - A subárvore para computar 4 ocorre 3 vezes
  - A subárvore para computar 3 ocorre 3 vezes
  - etc.
  - Para n grande, a qtde. de subárvores repetidas aumenta muito
- Vide animação: `https://visualgo.net/bn/recursion`

Exercício 2

Parte 1: solução 2

- Podemos melhorar o desempenho[5] se memoizarmos subproblemas que já foram computados
- Para memoizar, precisamos de uma estrutura de dados auxiliar com acesso eficiente (e.g. uma hashtable)

```
1  (define (fibonacci-memo n)
2    (define ht (make-hash))
3    (let fibonacci-recur ([n n])
4      (cond [(= n 1) 1]
5            [(= n 2) 1]
6            [(hash-has-key? ht n)
7             (hash-ref ht n)]
8            [else (let* ([a (fibonacci-recur (- n 1))]
9                         [b (fibonacci-recur (- n 2))]
10                        [c (+ a b)])
11                   (hash-set! ht n c)
12                   c)])))
```

```
> (time (fibonacci-memo 40))
cpu time: 1 real time: 0 gc time: 0
102334155
```

- Desvantagens dessa solução:
  - Código imperativo para manutenção da estrutura de dados
  - Manutenção da cache "poluindo" o algoritmo

[5] A versão memoizada é $O(n)$ até memoizar os valores, somando-se um outro custo posteriormente, correspondente às consultas na hashtable. Usa $O(n)$ em espaço para manter a cache.

Exercício 2

Parte 1: solução 3

- Há bibliotecas que permitem a memoização transparente de uma função recursiva:

```
raco pkg install memoize
```

```
1 (require memoize)
2
3 (define/memo (fibonacci-recur-memo n)
4   (cond [(= n 1) 1]
5         [(= n 2) 1]
6         [else (+ (fibonacci-recur-memo (- n 1))
7                  (fibonacci-recur-memo (- n 2)))]))
```

```
> (time (fibonacci-recur-memo 40))
cpu time: 0 real time: 0 gc time: 0
102334155
```

  - Não seria difícil definir o nosso próprio wrapper memoizador
  - Porém, precisaríamos saber funções de alta ordem e, talvez, macros, que são assuntos de aulas futuras
  - Pergunta: funções memoizadas mantêm a transparência referencial?

Atenção: a memoização não transforma a recursão dupla em recursão simples. O que acontece é que a recursão em árvore torna-se uma recursão em grafo acíclico direcionado (i.e. cria-se novas arestas para valores que já foram computados).

Exercício 2

Parte 2

- Ao invés de hahstable, podíamos pensar na memoização via uma tabela ou um vetor
- Nesse vetor, são armazenados os valores a cada chamada recursiva:

```
[1 1]
[1 1 2]
[1 1 2 3]
[1 1 2 3 5]
[1 1 2 3 5 8]
[1 1 2 3 5 8 13]
[1 1 2 3 5 8 13 21]
```

- É fácil perceber que para gerar um novo valor (azul) só usamos os dois últimos valores computados (vermelho)
- Logo, não é eficiente manter na memória valores antigos
- Podemos pensar numa solução que só mantenha dois valores como “cache”

Exercício 2

Parte 2

- Versão iterativa[6] com recursão na cauda (`a` e `b` formam nossa cache):

```
1 (define (fibonacci n)
2   (if (< n 1)
3       (raise-argument-error 'factorial "positive integer" n)
4       (fibonacci-iter n 1 0 1)))
5
6 (define (fibonacci-iter n i a b)
7   (if (= i n)
8       b
9       (fibonacci-iter n (+ i 1) b (+ a b))))
```

```
> (time (fibonacci 40))
cpu time: 0 real time: 0 gc time: 0
102334155
```

- Em suma, a versão iterativa pode ser vista como um caso extremo da versão memoizada explicitamente

[6] A versão iterativa é $O(n)$ em tempo e $O(1)$ em espaço.

Exercício 3

Construa um procedimento que teste se um número primo.

Exercício 3

Solução 1

- Testa praticamente todos os antecessores como divisores:

```
1  (define (divides a b)
2    (= (modulo a b) 0))
3
4  (define (prime? n)
5    (if (< n 2)
6        #f
7        (prime-f? n 2)))
8
9  ; prime  test with exhaustive divisor search
10 (define (prime-f? n i)
11   (cond [(= n i) #t]
12         [(divides n i) #f]
13         [else (prime-f? n (+ i 1))]))
```

- Para números grandes, o teste torna-se inviável:

```
> (time (prime? 2147483647))
cpu time: 36347 real time: 36332 gc time: 0
#t
```

Exercício 3

Solução 2

- Podemos diminuir o espaço de busca se usarmos a propriedade:
  Todo número composto tem um fator que é menor ou igual à sua raiz quadrada[7]

```
1 (define (prime-a? n)
2   (let prime-a-i ([i 2])
3     (cond [(> (* i i) n) #t]
4           [(divides n i) #f]
5           [else (prime-a-i (+ i 1))])))
```

[7] Se n é composto, é possível fatorá-lo como $n = ab$, sendo $a > 1$ e $b < n$. Se $a > \sqrt{n}$ e $b > \sqrt{n}$, então $n = \sqrt{n} \cdot \sqrt{n} < a \cdot b = n$, que é uma contradição.

Exercício 3

Solução 3

- Outra propriedade interessamte:
  Se o número não é divisível por 2, também não será divisível por qualquer múltiplo de 2
  Com isso, podemos eliminar todos os números pares do nosso teste

```
1 (define (prime-b? n)
2   (cond [(= n 2) #t]
3         [(divides n 2) #f]
4         [else (let prime-b-i ([i 3])
5                 (cond [(> (* i i) n) #t]
6                       [(divides n i) #f]
7                       [else (prime-b-i (+ i 2))]))]))
```

Exercício 3

Soluções 1, 2 e 3: medição de tempo de execução

```
> (time (prime? 6700417))
cpu time: 117 real time: 118 gc time: 0
#t
>
  (time (prime-a? 6700417))
cpu time: 0 real time: 0 gc time: 0
#t
> (time (prime-a? 67280421310721))
cpu time: 145 real time: 145 gc time: 0
#t
> (time (prime-a? 2147483647))
cpu time: 1 real time: 0 gc time: 0
#t
>
  (time (prime-b? 6700417))
cpu time: 1 real time: 0 gc time: 0
#t
> (time (prime-b? 67280421310721))
cpu time: 73 real time: 73 gc time: 0
#t
> (time (prime-b? 2147483647))
cpu time: 1 real time: 1 gc time: 0
#t
```

Exercícios para entregar

Exercício 4

Construa um procedimento que aproxime a série de Taylor para o seno de um ângulo x:

$$seno(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

O procedimento deverá apresentar recursão na cauda e deverá evitar recálculo de subproblemas sobrepostos. O erro de aproximação deve ser um parâmetro da função:

```
(seno 2 0.0001)
```

Exercício 5

O seguinte padrão de números é denominado triângulo de Pascal:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
        ....
```

Os números nas bordas do triângulo são todos 1, e cada número dentro do triângulo é a soma dos dois números acima dele. Podemos definir a função $T(i, j)$ que retorna o j-ésimo elemento da i-ésima linha do triângulo de Pascal. Por exemplo, $T(0, 0) = 1$ e $T(3, 2) = 3$.

1. Proponha uma definição recursiva a função $T(i, j)$ e implemente a função com base nessa definição.
2. Analise o trace da função e ateste a ocorrência de recursão em árvore. Proponha uma otimização via memoização e teste o desempenho.
3. O elemento $T(i, j)$ do triângulo de Pascal coincide com a quantidade de combinações de i elementos amostrados de j em j, i.e. $T(i, j) = C(i, j) = \binom{i}{j} = \frac{i!}{(i-j)!j!}$. Proponha uma solução iterativa com base nessa definição. Proponha uma otimização para evitar cálculos sobrepostos nos fatoriais.

Exercício 6

No teste de primalidade, podemos diminuir ainda mais o espaço de busca por divisores se considerarmos a seguinte propriedade:

Todo número primo, exceto 2 e 3, tem a forma $6k \pm 1$ *e todo número composto tem um fator primo.*

Consequentemente, caso n não seja 2 ou 3, nem seja divisível por 2 ou 3, basta usarmos como divisores os primos gerados por essa sequência, considerando também a propriedade que se n é composto, então tem um divisor menor ou igual que $\sqrt{n}$. Um exemplo de expansão da sequência é 5, 7, 11, 13, 17, 19, 23, 25,...

1. Construa uma função que teste a primalidade com base nessa propriedade.
2. Faça testes de tempo de execução comparando sua implementação com as vistas anteriormente, considerando inteiros grandes.

Recursos recomendados

- Modules. Racket Guide. URL:
  `https://docs.racket-lang.org/guide/modules.html`
- RackUnit. URL:
  `https://docs.racket-lang.org/rackunit/`

Funcional II: recursão e iteração

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

26 de junho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

Crédito de parte das imagens, a menos se especificado: Wikipedia