Funcional IV: listas e funções de ordem superior

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

Objetivos

- TODO

Parte 1: primalidade sem listas

Exercício 1

Construa um procedimento que teste se um número primo.

Exercício 1

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 1

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[1]

```
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))])))
```

[1] 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 1

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 1

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
```

Exercício 2

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

- Todo número primo, exceto 2 e 3, tem a forma $6k \pm 1$ [2]
- 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 nessas propriedades
2. Faça testes de tempo de execução comparando sua implementação com as vistas anteriormente, considerando inteiros grandes

[2] Atenção: isso não implica que todo número $6k \pm 1$ é primo.

Exercício 2: solução

Solução

```
1  (define (prime? n)
2    (prime-c? n))
3
4  (define (prime-c? n)
5    (cond [(= n 2) #t]
6          [(= n 3) #t]
7          [else (let loop ([k 1] [i 2] [j 3])
8                  (cond [(> (* i i) n) #t]
9                        [(divides n i) #f]
10                       [(divides n j) #f]
11                       [else (loop
12                               (+ k 1)
13                               (- (* 6 k) 1)
14                               (+ (* 6 k) 1))]))]))
```

Exercício 2: solução

Teste de desempenho

```
 1 (time (prime-a? 6700417))
 2 (time (prime-a? 67280421310721))
 3 (time (prime-a? 2147483647))
 4
 5 (display "\n")
 6
 7 (time (prime-b? 6700417))
 8 (time (prime-b? 67280421310721))
 9 (time (prime-b? 2147483647))
10
11 (display "\n")
12
13 (time (prime-c? 6700417))
14 (time (prime-c? 67280421310721))
15 (time (prime-c? 2147483647))
```

```
$ racket primes.rkt
cpu time: 0 real time: 0 gc time: 0
#t
cpu time: 147 real time: 147 gc time: 0
#t
cpu time: 1 real time: 1 gc time: 0
#t

cpu time: 0 real time: 0 gc time: 0
#t
cpu time: 76 real time: 75 gc time: 0
#t
cpu time: 0 real time: 1 gc time: 0
#t

cpu time: 0 real time: 0 gc time: 0
#t
cpu time: 27 real time: 27 gc time: 0
#t
cpu time: 0 real time: 0 gc time: 0
#t
```

Exercício 2

Testes

```
1  ; primes in [2,200]
2  (define primes '(2 3 5 7 11
3                   13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
4                   101 103 107 109 113 127 131 137 139 149 151 157 163 167 173
5                   179 181 191 193 197 199))
6  ; composites in [1,200]
7  (define composites '(4 6 8 9
8                       12 14 15 16 18 20 21 22 24 25 26 27 28 30 32 33 34 35 36
9                       38 39 40 42 44 45 46 48 49 50 51 52 54 55 56 57 58 60 62
10                      63 64 65 66 68 69 70 72 74 75 76 77 78 80 81 82 84 85 86
11                      87 88 90 91 92 93 94 95 96 98 99
12                      100 102 104 105 106 108 110 111 112 114 115 116 117 118
13                      119 120 121 122 123 124 125 126 128 129 130 132 133 134
14                      135 136 138 140 141 142 143 144 145 146 147 148 150 152
15                      153 154 155 156 158 159 160 161 162 164 165 166 168 169
16                      170 171 172 174 175 176 177 178 180 182 183 184 185 186
17                      187 188 189 190 192 194 195 196 198 200))
18 (define primes-suite
19   (test-suite
20     "Test primality related operations"
21     (check-true  (andmap prime? primes))
22     (check-false (ormap prime? composites))
23 (run-tests primes-suite)
```

```
$ racket primes-tests.rkt
2 success(es) 0 failure(s) 0 error(s) 2 test(s) run
```

Parte 2: primalidade com listas

Exercício 3

Construa o procedimento `(gen-primes n)` cujo valor é uma lista com todos os números primos em $[2, n]$.

Exemplo de chamada:

```
> (gen-primes 20)
'(2 3 5 7 11 13 17 19)
```

Exercício 3

Solução 1[3]: testes baseados em divisores ímpares com geração de listas intermediárias

```
1  (define (divisivel? a b)
2    (= (modulo a b) 0))
3
4  (define (possiveis-divisores a)
5    (cons 2
6      (cdr (build-list (floor (/ a 2))
7       (lambda(x) (add1 (* x 2)))))))
8
9  (define (primo? n)
10   (or (= n 2) (and
11    (> n 2)
12    (not (ormap
13      (lambda (x) (divisivel? n x))
14      (possiveis-divisores n))))))
15
16 (define (primos-ate n)
17   (filter primo? (build-list (sub1 n) (lambda (x) (+ 2 x)))))
```

[3] Código-fonte correspondente à semana 2

Exercício 3

Solução 1 — testes de propriedades e desempenho

```
1 (define prop-primos-ate
2   (property
3    ([n (choose-integer 2 (random 10000))])
4    (andmap primo? (primos-ate n))))
```

```
> (primos-ate 40)
'(2 3 5 7 11 13 17 19 23 29 31 37)
```

```
$ racket primos.rkt
OK, passed 100 tests.
```

```
> (time (let () (primos-ate 10000) (void)))
cpu time: 612 real time: 611 gc time: 10
```

Exercício 3

Solução 2: tentativas de divisão exaustivas

- Vamos diminuir o uso de memória evitando a geração de listas intermediárias:

```
1 (define (gen-primes-a n)
2   (let gen-primes-a-i ([i 2] [p null])
3     (cond [(> i n) (reverse p)]
4           [(prime? i) (gen-primes-a-i (+ i 1) (cons i p))]
5           [else (gen-primes-a-i (+ i 1) p)])))
```

```
> (gen-primes-a 20)
'(2 3 5 7 11 13 17 19)
```

- Porém, estamos gerando divisores sequenciais, é possível melhorar

Exercício 3

Solução 3: Usando propriedade $6k \pm 1$

- A propriedade $6k \pm 1$ gera menos divisores do que o conjunto dos possíveis divisores ímpares
- Adicionalmente, gera números utilizando um padrão regular:

```
1 (define (gen-pair k)
2   (list (- (* 6 k) 1) (+ (* 6 k) 1)))
3
4 (define (gen-wheel-pairs n)
5   (let rec ([k 1])
6     (if (= k n)
7         null
8         (cons (gen-pair k) (rec (+ k 1))))))
```

```
> (gen-wheel-pairs 10)
'((5 7) (11 13) (17 19) (23 25) (29 31) (35 37) (41 43) (47 49) (53 55))
```

- Para cada par (i_k, j_k):
  $j_k - i_k = 2$
  $i_{k+1} - j_k = 4$

Exercício 3

Solução 2: Usando propriedade $6k \pm 1$

- Algoritmo usando as razões do slide anterior:

```
1  ; cons in primes only if i is prime
2  (define (cons-prime i primes)
3    (if (prime? i)
4        (cons i primes)
5        primes))
6
7  ; prime enumeration based on 6k+-1 property
8  (define (gen-primes-b n)
9    (cond [(< n 2) null]
10         [(= n 2) (list 2)]
11         [(= n 3) (list 2 3)]
12         [else (let loop ([i 5] [k 2] [primes '(3 2)])
13                 (cond [(> i n) (reverse primes)]
14                       [(= k 2) (loop (+ i k) 4 (cons-prime i primes))]
15                       [else (loop (+ i k) 2 (cons-prime i primes))]))]))
```

```
> (gen-primes-b 20)
'(2 3 5 7 11 13 17 19)
```

Exercício 3

Soluções 2 e 3: testes

```
1  (require rackunit "primes.rkt")
2  (require rackunit/text-ui)
3
4  ; primes in [2,200]
5  (define primes '(2 3 5 7 11
6                   13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
7                   101 103 107 109 113 127 131 137 139 149 151 157 163 167 173
8                   179 181 191 193 197 199))
9  ; composites in [1,200]
10 (define composites '(4 6 8 9
11                      12 14 15 16 18 20 21 22 24 25 26 27 28 30 32 33 34 35 36
12                      38 39 40 42 44 45 46 48 49 50 51 52 54 55 56 57 58 60 62
13                      63 64 65 66 68 69 70 72 74 75 76 77 78 80 81 82 84 85 86
14                      87 88 90 91 92 93 94 95 96 98 99
15                      100 102 104 105 106 108 110 111 112 114 115 116 117 118
16                      119 120 121 122 123 124 125 126 128 129 130 132 133 134
17                      135 136 138 140 141 142 143 144 145 146 147 148 150 152
18                      153 154 155 156 158 159 160 161 162 164 165 166 168 169
19                      170 171 172 174 175 176 177 178 180 182 183 184 185 186
20                      187 188 189 190 192 194 195 196 198 200))
21
22 (define primes-suite
23   (test-suite
24     "Test with a list of primes"
25     (check-true  (andmap prime? primes))
26     (check-false (ormap prime? composites))
27     (check-true  (equal? (gen-primes-a 200) primes))
28     (check-true  (equal? (gen-primes-b 200) primes))))
29
30 (run-tests primes-suite)
```

```
$ racket primes-tests.rkt
4 success(es) 0 failure(s) 0 error(s) 4 test(s) run
```

Exercício 3

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

```
> (time (let () (gen-primes-a 10000) (void)))
cpu time: 3 real time: 3 gc time: 0
> (time (let () (gen-primes-b 10000) (void)))
cpu time: 3 real time: 3 gc time: 0
> (time (let () (primos-ate 10000) (void)))
cpu time: 612 real time: 611 gc time: 10

> (time (let () (gen-primes-a 6700417) (void)))
cpu time: 6200 real time: 6197 gc time: 19
> (time (let () (gen-primes-b 6700417) (void)))
cpu time: 5971 real time: 5969 gc time: 21
```

- **Exercício 4**: Por quê as soluções 2 e 3 diferem pouco no tempo de execução? Justifique com base na análise do algoritmo ou com exemplos de traço de execução.

Exercício 5

Construa um procedimento que fatore um inteiro positivo fornecido como parâmetro. O valor do procedimento deve ser uma lista de todos os fatores.

Exemplo:

```
> (factorize 123)
'(3 41)
> (factorize 438)
'(2 3 73)
> (factorize 428)
'(2 2 107)
> (factorize 448)
'(2 2 2 2 2 2 7)
```

Exercício 5

Solução

```
1 | (define (factorize n)
2 |   (let loop ([n n] [factors null] [primes (gen-primes (sqrt n))])
3 |     (cond [(< n 2) (reverse factors)]
4 |           [(null? primes) (reverse (cons n factors))]
5 |           [(divides n (car primes))
6 |            (loop (quotient n (car primes)) (cons (car primes) factors) primes)]
7 |           [else (loop n factors (cdr primes))])))
```

Exercício 5

Testes

```
(define factors '( (5 . (5))
                   (...)
                   (255 . (3 5 17))
                   (256 . (2 2 2 2 2 2 2 2))
                   (257 . (257))
                   (258 . (2 3 43))
                   (259 . (7 37))
                   (...)
                   (776 . (2 2 2 97))
                   (777 . (3 7 37))
                   (778 . (2 389))
                   (779 . (19 41))
                   (780 . (2 2 3 5 13))
                   (781 . (11 71))))
(define primes-suite
  (test-suite
    "Test primality related operations"
    (...)
    (test-case
      "Test prime factoring"
      (define (test-factors pair)
        (equal? (factorize (car pair)) (cdr pair)))
      (check-true (andmap test-factors factors)))
    (...)))

(run-tests primes-suite)
```

```
$ racket primes-tests.rkt
5 success(es) 0 failure(s) 0 error(s) 5 test(s) run
```

Exercícios para entregar

Exercício 8

Com base na solução 1 para enumeração de primos, resolva os itens a seguir.

6. Crie testes utilizando o QuickCheck para verificar as funções `divisível?` e `possiveisDivisores`. Além disto, crie um teste adicional para a função `primos-ate` que verifica se a função obedece o parâmetro n, ou seja, que não são devolvidos primos maiores que n.
7. O n-ésimo número de Mersenne é dado pela fórmula $M_n = 2^n - 1$. Desde a antiguidade sabe-se que M_2, M_3, M_5 e M_7 são primos. Como 2, 3, 5 e 7 são primos, conjecturou-se que para todo p primo, **então** M_p também seria primo. Utilizando o QuickCheck mostre que este não é o caso.

Exercício 9

O máximo fator comum ou máximo divisor comum (mdc) de um dois ou mais números é o produto de todos os fatores primos comuns a esses números.

1. Construa `(mdc x y)` com base nessa propriedade. Você pode usar a função `factorize`. Pode usar também a função `lset-intersection` da `srfi/1`.
2. Ao aplicar `factorize` consecutivamente para dois números, cada aplicação gera uma nova lista de primos, porém as duas listas geradas apresentarão intersecção potencialmente grande se os números forem próximos (i.e. gero primos repetidos desnecessariamente). Por exemplo, `mdc(68,156) => 4`, mas `(gen-primes (sqrt 68)) => '(2 3 5 7)` e `(gen-primes (sqrt 156)) = '(2 3 5 7 11)`. Trate esse problema gerando uma única lista de primos, apenas para o maior dos números.
3. Outra estratégia para calcular mdc é o algoritmo de Euclides, que tem a seguinte definição iterativa: se x é maior que y, então se y divide x, então o mdc de x e y é y; caso contrário, o mdc de x e y é o mesmo que o mdc de `(modulo` x y`)` e y. Implemente o algoritmo de Euclides e compare o desempenho (em tempo de execução) com o algoritmo do item anterior.
4. Modifique o algoritmo de Euclides para que receba uma lista com uma quantidade arbitrária de números e determine o mdc entre esses números.

Exercício 10

O mínimo múltiplo comum (mmc) de uma lista de números pode ser computado pela divisão sucessiva dos elementos da lista pela sequência de números primos, sempre que seja possível dividir um dos elementos da lista por um primo, até que todos os elementos da lista sejam 1. O mmc é produto de todos os divisores primos utilizados.
Exemplo:

```
(4 7 12 21 42)      2
(2 7 6  21 21)      2
(1 7 3  21 21)      3
(1 7 1  7  7)       7
(1 1 1  1  1)

mmc(4,7,12,21,42) = 2 * 2 * 3 * 7 = 84
```

1. Construa uma função que determine o mmc pelo método descrito acima.
2. Uma definição alternativa de mmc é estabelecida pela fórmula $mmc(a,b) = \frac{a \cdot b}{mdc(a,b)}$. Utilize o algoritmo de Euclides para construir uma função que determine o mmc de acordo com a fórmula. Compare o desempenho com a solução do item anterior.

Recursos recomendados

- Pairs and Lists. Racket Guide. URL: `https://docs.racket-lang.org/guide/pairs.html`

Funcional IV: listas e funções de ordem superior

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