Exercícios da cadeira de Introdução à Programação

Cláudia Antunes
Ana Cardoso Cachopo
João Cachopo
Francisco Couto
António Leitão
Inês Lynce
César Pimentel
H. Sofia Pinto

Ano Lectivo 2002/2003
Parte II

Conteúdo

7 Programação imperativa

Exercício 7.1
Diga o que é impresso pelo interpretador de Scheme ao avaliar cada uma das seguintes expressões. Suponha que as expressões são avaliadas pela ordem apresentada. Assinale e explique os erros detectados.

```
(+ 1 2.0)

(first '(+ (sqrt 5) 4))

odd?

(define x 5)

(set! x "ip")

(set! b 2)

(list x 'b 5 (* 3 2))

(define a 3)

(set! a "ola")

(+ a 1)

(begin
  (let ((a 5))
    (+ a (* 45 327))
    (sqrt (length '(1 a b "bom dia" (2 5) 3))))
  (display 'a)
  a)
```

Exercício 7.2
Escreva um procedimento chamado `cria-multiplicador` que recebe um número e retorna um procedimento que recebe outro número e o multiplica pelo primeiro. Por exemplo,

```
> (define mult5 (cria-multiplicador 5))
> (mult5 2)
10
```

Exercício 7.3
(Livro — 3.1) Um acumulador é um procedimento que é chamado repetidamente com apenas um argumento numérico e acumula os seus argumentos numa soma. De cada vez que é chamado, retorna a soma acumulada até ao momento.
Escreva um procedimento `make-accumulator` que gera acumuladores, cada um dos quais mantendo uma soma independente. O valor de entrada para o procedimento `make-accumulator` deve especificar o valor inicial da soma. Por exemplo,

```
(define A (make-accumulator 5))

(A 10)
15

(A 10)
25
```

Exercício 7.4
(Livro — 3.2) Em aplicações para testar software, é útil ser capaz de contar o número de vezes que um procedimento é chamado durante o decurso de uma computação.
Escreva um procedimento `make-monitored` que recebe um procedimento `f` como argumento, que por sua vez é um procedimento de um argumento. O resultado retornado pelo procedimento `make-monitored` é um terceiro procedimento `mf` que mantém um registo do número de vezes que foi chamado através de um contador interno. Se o valor de entrada para `mf` for o símbolo `how-many-calls?`, então `mf` deve retornar o valor do contador. Se o valor de entrada for o símbolo `reset-count`, então `mf` deve inicializar o contador a zero. Para qualquer outro valor de entrada, `mf` retorna o valor de aplicar `f` a esse valor e incrementa o contador. Por exemplo, podemos criar uma versão monitorizada do procedimento `sqrt`:

```
(define s (make-monitored sqrt))

(s 100)
10

(s 'how-many-calls?)
1
```

Exercício 7.5
(Livro — 3.27) Como foi discutido nas aulas teóricas, o seguinte procedimento para calcular os números de *Fibonacci* não é eficiente porque calcula múltiplas vezes o mesmo valor:

```
(define (fib n)
   (cond ((= n 0) 0)
         ((= n 1) 1)
         (else (+ (fib (- n 1))(fib (- n 2))))))
```

Para evitar a repetição de cálculos, podemos imaginar um procedimento com estado interno que vai "aprendendo"e memorizando os números de Fibonacci, à medida que estes são calculados pelo procedimento. Este procedimento mantém uma lista com os valores dos números de Fibonacci que já memorizou, por exemplo `((0 . 0)(1 . 1)(2 . 1))`. Sempre que tem de calcular um número de Fibonacci, começa por ver se já sabe o seu valor, e se não souber calcula-o e memoriza-o. Escreva um procedimento em Scheme para calcular os números de Fibonacci de acordo com esta técnica.

Exercício 7.6
Defina um procedimento `vector-search` que recebe um número e um vector e devolve a posição no vector onde esse número ocorre, ou o valor lógico falso no caso de o número não existir no vector.

```
> (vector-search 1 '#(1 2 3 4 5))
0
> (vector-search 3 '#(1 2 3 4 5))
2
> (vector-search 8 '#(1 2 3 4 5))
#f
```

Exercício 7.7
Defina um procedimento `vector-map` que corresponde ao procedimento `map`, mas para vectores: recebe um procedimento de um argumento e um vector e retorna um vector com os resultados produzidos ao aplicar o procedimento a cada elemento do vector recebido.

```
> (vector-map abs '#(-10 2.5 -11.6 17))
#(10 2.5 11.6 17)

> (define v '#(1 2 3))
> (vector-map (lambda (x) (* x x)) v)
#(1 4 9)
> v
#(1 2 3)
```

Exercício 7.8
Defina um procedimento `vector-map-into!`, semelhante ao procedimento `vector-map` do exercício anterior, mas que altera destrutivamente o vector recebido subtituindo os elementos do vector pelo resultado de lhes aplicar o procedimento recebido.

```
> (vector-map-into! abs '#(-10 2.5 -11.6 17))
#(10 2.5 11.6 17)

> (define v '#(1 2 3))
> (vector-map-into! (lambda (x) (* x x)) v)
#(1 4 9)
> v
#(1 4 9)
```

Exercício 7.9
Defina um procedimento `vector-reverse!` que recebe um vector e inverte esse vector, modificando-o destrutivamente.

```
> (vector-reverse! '#(1 2 3 4 5))
#(5 4 3 2 1)

> (define v '#(1 2 3))
> (vector-reverse! v)
#(3 2 1)
> v
#(3 2 1)
```

Exercício 7.10
O procedimento `vector-search` do exercício 7.6 tem que percorrer todos os elementos do vector no caso de estar a procurar um elemento que não existe no vector.

1. No caso de o elemento a procurar existir no vector, quantos elementos do vector, em média, serão percorridos?

2. Supondo que o vector está ordenado por ordem crescente, defina um nova versão do procedimento `vector-search` que efectua uma procura sequencial mas que não tenha que percorrer todos os elementos do vector quando o elemento a procurar não existe no vector.

3. Em média, quantos elementos do vector são percorridos quando o número não existe no vector? E quando existe?

Exercício 7.11
Sabendo que o vector está ordenado por ordem crescente, é possível realizar uma procura binária, que é mais eficiente que a procura sequencial do exercício anterior. Esta procura baseia-se no facto de que se compararmos o elemento a procurar com o elemento do meio do vector podemos obter um de três resultados diferentes:

- Os dois elementos são iguais, o que significa que encontrámos o elemento que procurávamos.

- O elemento do meio é menor, o que significa que o elemento que procuramos, se existir, estará na metade do vector com índices maiores.

- O elemento do meio é maior, o que significa que o elemento que procuramos, se existir, estará na metade do vector com índices menores.

Defina o procedimento `vector-binary-search` que realiza uma procura binária num vector. Qual a ordem de crescimento deste procedimento?

8 Avaliação baseada em ambientes

Exercício 8.1
Desenhe o diagrama de ambientes criado pela seguinte interacção:

```
(define x 63)

(define square
  (lambda (x)
    (* x x)))

(define sum-sq
  (lambda (x y)
    (+ (square x) (square y))))

(sum-sq 3 4)
```

Exercício 8.2
Desenhe o diagrama de ambientes criado pela seguinte interacção:

```
> (define (conjuncao p1 p2)
    (lambda (x)(and (p1 x)(p2 x))))
> (define par? even?)
> (define par>3?
    (conjuncao par? (lambda (x)(> x 3))))
> (par>3? 4)
#t
> (par>3? 2)
#f
> (par>3? 5)
#f
```

Exercício 8.3
Desenhe o diagrama de ambientes criado pela seguinte interacção:

```
> (define (cria-acumulador valor)
    (lambda (increm)
      (set! valor (+ valor increm))
      valor))
> (define ac1 (cria-acumulador 0))
> (ac1 5)
5
> (ac1 5)
10
```

Exercício 8.4
Desenhe o diagrama de ambientes criado pela seguinte interacção:

```
> (define (cria-acumulador vi)
    (let ((valor vi))
      (lambda (x)
        (set! valor (+ valor x)))))
> (define A (cria-acumulador 0))
> (A 5)
```

Exercício 8.5
Desenhe o diagrama de ambientes criado pela seguinte interacção:

```
> (define (cria-multiplicador n)
    (lambda (x)
      (* x n)))
> (define mult3 (cria-multiplicador 3))
> (define mult5 (cria-multiplicador 5))
> (mult3 2)
6
> (mult5 7)
35
```

Exercício 8.6
Considere os seguintes exemplos de código:

Exemplo A:

```
> (define (procA1 x y)
    (define (procA2 x)
      (define (procA3 y)
        (+ x y))
      (procA3 x))
    (procA2 y))

> (procA1 2 3)
```

Exemplo B:

```
> (define (procB1 x y)
    (define (procB2 x)
      (procB3 x))
    (define (procB3 y)
      (+ x y))
    (procB2 y))

> (procB1 2 3)
```

Desenhe os diagramas de ambientes criados por cada um e mostre de forma clara o resultado final produzido.

Exercício 8.7
Desenhe o diagrama de ambientes criados pela seguinte interacção:

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(define addthree (make-adder 3))

(define addfive (make-adder 5))

(addfive 7)

(addthree 7)
```

Exercício 8.8
Usando os diagramas de ambientes explique a diferença existente entre as duas expressões:

Ex. A

```
(let ((x 1)
      (y 2))
  (let ((x 4)
        (z (+ x 4)))
    (set! y (+ x z))
    (display (+ x y z)))
  (+ x y))
```

Ex. B

```
(let ((x 1)
      (y 2))
  (let ((x 4)
        (z (+ x 4)))
    (define y (+ x z))
    (display (+ x y z)))
  (+ x y))
```

Exercício 8.9
Os ambientes permitem-nos perceber como é que podemos usar procedimentos como representações para tipos abstractos de dados. Por exemplo, podemos criar rectângulos da seguinte forma:

```
(define (make-rect w h)
  (define (dispatch op)
    (cond ((eq? op 'width) w)
          ((eq? op 'height) h)
          ((eq? op 'area) (* w h))
          ((eq? op 'perimeter) (* 2 (+ w h)))
          (else (error "rectangle: non-existent operation" op))))
  dispatch)

(define r1 (make-rect 5 30))

(r1 'height)
```

Desenhe o diagrama de ambientes criados pelo código acima.

9 Estruturas mutáveis

Exercício 9.1
Diga qual o resultado produzido pelo avaliador de Scheme quando as seguintes expressões são avaliadas sequencialmente. Desenhe o diagrama de caixas e ponteiros correspondentes. Identifique o lixo que é recolhido pelo mecanismo de *garbage collection*.

1. ```
   (define a (cons 3 ()))
   (set! a (cons 4 (cons 5 ())))
   (define a (cons 1 (cons 2 ())))
   (define b (cons (cons 3 ())(cons 4 (cons 5 ()))))
   (define c (cons a (cdr b)))
   (set-car! b a)
   (equal? b c)
   (eq? b c)
   ```

2. ```
   (define c (cons 1 (cons 2 (cons 3 ()))))
   (define d (cons 4 (cons 5 (cons 6 ()))))
   (define w (cons c d))
   (eq? (car w) c)
   (set-cdr! c d)
   w
   ```

Exercício 9.2

1. Defina o procedimento `append`, de modo a que gere um processo recursivo.
2. Defina o procedimento `append!`, que se distingue do anterior pelo facto de não fazer cópia de estruturas. (Sugestão: comece por definir o procedimento `last-pair`.)
3. Qual o resultado / diagrama de caixas e ponteiros produzido pela avaliação das seguintes expressões?

   ```
   (define c (cons 1 (cons 2 (cons 3 ()))))
   (define d (cons 4 (cons 5 (cons 6 ()))))
   (define u (cons c (cons d ())))
   u
   (define x (append c d))
   (set-cdr! c '(7))
   (set-cdr! d '(8))
   x
   (define y (append! c d))
   y
   u
   ```

Exercício 9.3
Desenhe o diagrama de ambientes (com caixas e ponteiros) criado pelo código que se segue:

```
(let ((x (list 1 2 3)))
  (let ((y (list 4 5 6)))
    (let ((z (cons x y)))
      (set-cdr! x y)
      z)))
```

Exercício 9.4
Defina o procedimento map!, que tem um comportamento semelhante ao procedimento map, mas altera destrutivamente a lista recebida como argumento.

```
> (define l (list 1 2 3))
> (map add1 l)
(2 3 4)
> l
(1 2 3)
> (map! add1 l)
(2 3 4)
> l
(2 3 4)
```

Exercício 9.5
Desenhe o diagrama de caixas e ponteiros produzido pela avaliação do código que se segue:

```
(let ((x (cons 1 1)))
  (set-car! x x)
  (set-cdr! x x)
  x)
```

Qual o valor devolvido pela avaliação da expressão?

Exercício 9.6
Explique o que acontece com a avaliação do código que se segue:

```
(define (mistery x)
  (let ((b (last-pair x)))
    (set-cdr! b x)
    x))

(define answer (mistery (list 1 2 3)))

(print-elements answer)
```

Considere que o procedimento `print-elements` está definido do seguinte modo:

```
(define (print-elements l)
  (if (null? l)
      (print)
      (begin
        (print (first l))
        (print-elements (rest l)))))
```

Sugira soluções para este problema.

10 Programação com objectos

Exercício 10.1
Um `contador` é um procedimento com estado interno que mantém um valor de contagem, e que recebe como argumento um dos seguinte símbolos: *incrementa, decrementa, valor-actual, poe-a-zero*. O valor retornado pelo contador corresponde ao valor de contagem depois de efectuada a operação pedida: incremetar ou decrementar de uma unidade, devolver o valor actual sem o modificar e colocar o valor a zero, respectivamente. Se o argumento recebido não for nenhum destes símbolos, o contador assinala um erro. Escreva um procedimento que recebe o valor inicial de contagem de um contador e cria o contador correspondente. Por exemplo,

```
> (define c1 (cria-contador 5))
> (c1 'incrementa)
6
> (define c2 (cria-contador 10))
> (c2 'decrementa)
9
> (c1 'poe-a-zero)
0
> (c2 'valor-actual)
9
```

Exercício 10.2
Escreva um procedimento em Scheme chamado `cria-estacionamento` que simula o funcionamento de um parque de estacionamento. O procedimento recebe um inteiro que determina a lotação do parque e devolve um procedimento que pode receber como argumentos *entra-carro, sai-carro* e *num-lugares*. Por exemplo,

```
> (define estacionamento-IST (cria-estacionamento 15))
> (estacionamento-IST 'entra-carro)
> (estacionamento-IST 'entra-carro)
> (estacionamento-IST 'num-lugares)
13
> (estacionamento-IST 'sai-carro)
> (estacionamento-IST 'num-lugares)
14
```

Exercício 10.3
Escreva um procedimento chamado `faz-lanca-notas` que devolve um procedimento com estado interno que permite simular o lançamento de notas para um aluno a uma determinada cadeira. Este procedimento pode receber como argumento um inteiro ou um símbolo. Dependendo do argumento, tem o seguinte comportamento:

- Inteiro - atribui nota à cadeira (considere que este valor é sempre introduzido correctamente entre 0 e 20); no limite só podem ser lançadas duas notas positivas;
- `nota` - devolve a nota lançada (a melhor nota no caso de ter feito melhoria);

- `passou-cadeira?` - devolve um valor lógico consoante o aluno tenha passado ou não à cadeira.
- `fez-melhoria?` - devolve um valor lógico consoante o aluno tenha feito ou não melhoria de nota. Considera-se que um aluno fez melhoria de nota se já foram lançadas duas notas positivas.

Por exemplo,

```
> (define Aluno99999 (faz-lanca-notas))
> (Aluno99999 'passou-cadeira?)
#f
> (Aluno99999 14)
> (Aluno99999 'nota)
14
> (Aluno99999 17)
> (Aluno99999 'fez-melhoria?)
#t
> (Aluno99999 18)
"Lancamentos de nota esgotados"
```

Exercício 10.4
Defina uma classe que corresponde a uma urna de uma votação. A sua classe deve receber a lista dos possíveis candidatos e manter como estado interno o número de votantes em cada candidato. Esta classe pode receber um voto num dos possíveis candidatos, aumentando o número de votos desse candidato em um. Deve também permitir apresentar os resultados da votação. Por exemplo,

```
> (define urna (cria-urna '(A B C)))
> (urna 'resultados)
c:0 b:0 a:0
> (urna 'A)
> (urna 'C)
> (urna 'A)
> (urna 'B)
> (urna 'C)
> (urna 'A)
> (urna 'resultados)
c:2 b:1 a:3
```

Nota: Para apresenta os resultados, use o comando `display`.