

Exercícios da cadeira de Introdução à Programação

Ana Cardoso Cachopo
João Cachopo
António Leitão

Ano Lectivo 2000/2001

Prefácio

Este documento reúne os exercícios propostos nas aulas práticas da cadeira de Introdução à Programação da Licenciatura de Engenharia Informática e de Computadores do Instituto Superior Técnico nos anos lectivos 1998/1999 e 1999/2000. Esta compilação destina-se a ser usada na referida cadeira no ano lectivo 2000/2001.

Alguns destes exercícios correspondem à tradução de exercícios propostos no livro de texto referido na bibliografia da cadeira:

- Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996.

Sempre que tal se verifique aparece associado ao número do exercício uma indicação do número que o exercício tem no referido livro.

H. Sofia Pinto
(editora)

1 Série Nº1

Exercício 1.1

(Livro — 1.1 e outros) Em baixo é apresentada uma sequência de expressões. Diga qual é o resultado impresso pelo interpretador de Scheme quando é avaliada cada uma dessas expressões. Assuma que a sequência é avaliada pela ordem apresentada.

10

(+ 5 3 4)

(- 9 1)

(/ 6 2)

(+ (* 2 4) (- 4 6))

(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(+ (* (+ (* 2 3) 4) 5) 6)

(/ (* (/ (* 2 3) 3) 4) 4)

(+ (- (+ (- (+ 1 2) 3) 4) 5) 6)

(- (+ (- (+ (- 1 2) 3) 4) 5) 6)

Exercício 1.2

(Livro — 1.2) Traduza a seguinte expressão para a notação prefixa:

$$\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}$$

Exercício 1.3

Traduza a seguinte expressão para a notação prefixa:

$$5 + 4 + 3 + 2 + 1$$

Exercício 1.4

Traduza a seguinte expressão para a notação prefixa:

$$5^3 + 4^2$$

Exercício 1.5

Traduza a seguinte expressão para a notação prefixa:

$$(5 + 4)^3$$

Exercício 1.6

Traduza a seguinte expressão para a notação prefixa:

$$5 + 4 \cdot 3/2 + 1$$

Exercício 1.7

Traduza a seguinte expressão para a notação prefixa:

$$(5 + 4) \cdot 3/(2 + 1)$$

Exercício 1.8

Traduza a seguinte expressão para a notação prefixa:

$$(5 + 4 \cdot 3)/(2 + 1)$$

Exercício 1.9

Traduza a seguinte expressão para a notação prefixa:

$$5 + 4 \cdot (3/2 + 1)$$

Exercício 1.10

Traduza a seguinte expressão para a notação prefixa:

$$1 + 2 \cdot 3/4 - 5 \cdot 6/7 + 8$$

Exercício 1.11

Defina um procedimento que calcula o perímetro de uma circunferência $p = 2\pi r$.

Exercício 1.12

Defina um procedimento que calcula a área de uma circunferência $a = \pi r^2$.

Exercício 1.13

Defina um procedimento que calcula o volume de uma esfera $v = \frac{4}{3}\pi r^3$.

Exercício 1.14

Defina um procedimento que calcula o volume de uma casca esférica de raio interior r_1 e raio exterior r_2 .

Exercício 1.15

Defina um procedimento que calcula a altura h a que se encontra um objecto que se deixa cair de uma altura h_0 com velocidade inicial v_0 ao fim de um tempo t . ($h = h_0 - v_0 t - \frac{1}{2}gt^2$, $g = 9.8m/s^2$).

Exercício 1.16

Defina um procedimento chamado `hipotenusa` que, dados os comprimentos dos dois catetos de um triângulo rectângulo, calcula o comprimento da hipotenusa. Dados os comprimentos dos catetos a e b do triângulo, a hipotenusa h é calculada como: $h = \sqrt{a^2 + b^2}$

Exercício 1.17

Defina um procedimento chamado `perimetro` que, dados os comprimentos dos dois catetos de um triângulo rectângulo calcula o seu perímetro. O perímetro de uma figura geométrica é a soma dos comprimentos de todos os seus lados.

2 Série Nº2

Exercício 2.1

(Livro — 1.1) Em baixo é apresentada uma sequência de expressões. Diga qual é o resultado impresso pelo interpretador de Scheme quando é avaliada cada uma dessas expressões. Assuma que a sequência é avaliada pela ordem apresentada.

```
(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
       ((< a b) b)
       (else -1))
   (+ a 1))
```

Exercício 2.2

Defina o procedimento `max` que recebe dois números como argumentos e devolve o maior deles.

Exercício 2.3

Defina o procedimento `max3` que recebe três números como argumentos e devolve o maior deles.

Exercício 2.4

Defina o procedimento `factorial` que recebe um número como argumento e calcula o factorial desse número. Note que $n! = n \times (n - 1)!$ e $0! = 1$.

Exercício 2.5

Considere definidos os seguintes procedimentos: `add1`, `sub1` e `zero?`, que somam um ao seu argumento, subtraem um ao seu argumento, ou testam se o seu argumento é igual a zero, respectivamente.

Com base neles (ou seja, sem utilizar procedimentos como `+`, `-`, `*`, `/`, `=`, `<`, `>`), defina os seguintes procedimentos:

1. O procedimento `soma`, que recebe dois inteiros superiores ou iguais a zero x e y , e calcula a soma entre eles.
2. O procedimento `igual?`, que dados dois inteiros superiores ou iguais a zero x e y , retorna verdadeiro se eles forem iguais e falso caso contrário.
3. O procedimento `menor?`, que dados dois inteiros superiores ou iguais a zero x e y , indica se x é menor que y .
4. O procedimento `diferenca`, que calcula a diferença entre dois inteiros superiores ou iguais a zero x e y .
5. O procedimento `produto`, que calcula o produto entre dois inteiros superiores ou iguais a zero x e y . Para definir este procedimento pode também usar o procedimento `soma`.
6. O procedimento `divisao-inteira`, que calcula a divisão inteira entre dois inteiros positivos x e y . A divisão inteira entre x e y é o máximo inteiro d tal que $d \times y \leq x$. Para definir este procedimento pode usar os procedimentos `diferenca` e `menor?`.

Exercício 2.6

Escreva um procedimento para calcular o valor de $\text{sen}(x)$ utilizando a expansão em série:

$$\text{sen}(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

O seu procedimento deve ter procedimentos para calcular o factorial e a potência.

O seu procedimento deve receber, para além do valor em radianos para o qual quer calcular o seno, o número de termos que devem ser considerados.

Exercício 2.7

Considere uma versão do jogo do NIM em que dois jogadores jogam alternadamente. No início do jogo existe uma série de palitos. Em cada jogada, cada jogador pode remover 1, 2 ou 3 palitos. Quem remover o último palito ganha o jogo.

Defina os seguintes procedimentos para criar um programa que joga ao NIM:

1. O procedimento `ganha?`, que recebe como argumento o número de palitos ainda existentes. Este procedimento deve devolver o valor lógico verdadeiro se o jogador que vai jogar agora pode ganhar o jogo se jogar da melhor forma possível, e falso caso contrário. Sugestão: utilize o procedimento `perde?` da alínea seguinte.
2. O procedimento `perde?`, que recebe como argumento o número de palitos ainda existentes. Este procedimento deve devolver o valor lógico verdadeiro se o jogador que vai jogar agora vai perder o jogo se o adversário jogar da melhor forma possível, e falso caso contrário. Sugestão: utilize o procedimento `ganha?`.

3. O procedimento `nim`, que recebe como argumento o número de palitos ainda existentes e devolve o número de palitos que o jogador a jogar nessa situação deve remover para ganhar o jogo. Sugestão: utilize o procedimento `perde?`.

Exercício 2.8

(Livro — 1.4) Repare que o nosso modelo de avaliação permite a existência de combinações cujos operadores são expressões compostas. Use esta observação para descrever o comportamento do seguinte procedimento:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

Exercício 2.9

(Livro — 1.5) O Zé Só Bites inventou um teste para determinar se o interpretador com que ele se deparou usa avaliação pela ordem aplicativa ou avaliação pela ordem normal. Ele define os dois procedimentos:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

E depois avalia a expressão

```
(test 0 (p))
```

Qual é o comportamento que o Zé vai observar com um interpretador que use a ordem de avaliação aplicativa? Qual é o comportamento que ele vai observar com um interpretador que use a ordem de avaliação normal? Explique a sua resposta. (Assuma que a regra de avaliação para a forma especial `if` é a mesma, quer o interpretador use a ordem de avaliação aplicativa, quer use a ordem de avaliação normal: primeiro avalia o predicado e o resultado determina se deve avaliar o conseqüente ou a alternativa.)

Exercício 2.10

(Adaptado do Livro — 1.6) A Alice não percebe porque é que o `if` precisa de ser uma forma especial. “Porque é que não o posso definir como um procedimento normal em termos do `cond`?” pergunta ela. Eva, uma amiga sua, diz que isso é possível e define uma nova versão do `if`:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

A Eva demonstra o programa à Alice:

```
(new-if (= 2 3) 0 5)
5
```

```
(new-if (= 1 1) 0 5)
0
```

Encantada, a Alice usa o `new-if` para re-escrever o procedimento que calcula a potência:

```
(define (potencia base exp)
  (new-if (= exp 0)
          1
          (* base (potencia base (- exp 1)))))
```

O que é que acontece quando a Alice tenta calcular potências? Explique.

Exercício 2.11

(Livro — 1.3) Defina um procedimento que recebe três números como argumentos e devolve a soma dos quadrados dos dois maiores.

3 Série Nº3

Exercício 3.1

Diga qual o resultados das seguintes combinações:

((lambda (x y) (+ x y)) 1 2)

((lambda (f x y) (f x y)) * 2 3)

((lambda (* + x y z) (* x (+ y z))) + * 1 2 3)

Exercício 3.2

Defina um procedimento que calcula o factorial de n .

Modifique o procedimento anterior para que, no caso de n ser negativo, o factorial devolva zero.

Note que esse teste só deve ser feito uma vez.

Exercício 3.3

Defina um procedimento que calcula uma potência inteira de x . Note que $x^n = x * x^{n-1}$ e $x^0 = 1$.

Modifique o procedimento anterior para que passe também a conseguir calcular potências em que o expoente é negativo. Note que $x^{-n} = \frac{1}{x^n}$.

Exercício 3.4

Modifique o procedimento anterior para que passe a tratar correctamente os casos 0^n , com n negativo e 0^0 .

Exercício 3.5

Considere definidos os seguintes procedimentos: `add1`, `sub1` e `zero?`, que somam um ao seu argumento, subtraem um ao seu argumento, ou testam se o seu argumento é igual a zero, respectivamente.

Com base neles (ou seja, sem utilizar procedimentos como `+`, `-`, `*`, `/`, `=`, `<`, `>`), defina os seguintes procedimentos:

1. Defina o predicado `positivo?` que recebe um número qualquer (positivo ou negativo) e indica se ele é positivo.
2. Defina o procedimento `simetrico` que recebe um número qualquer (positivo ou negativo) e devolve o simétrico desse número.
3. Defina o teste de igualdade de dois números quaisquer (positivos ou negativos).
4. Defina o procedimento `divisao-inteira-positiva`, que calcula a divisão inteira entre dois inteiros positivos x e y . A divisão inteira entre x e y é o máximo inteiro d tal que $d \times y \leq x$. Para definir este procedimento pode usar os procedimentos `diferenca` e `menor?`.
5. Defina o procedimento `divisao-inteira` que calcula a divisão inteira entre dois números quaisquer (positivos ou negativos).

4 Série Nº4

Exercício 4.1

(Adaptado do Livro — 1.9) Cada um dos seguintes procedimentos define um método para adicionar dois inteiros positivos em termos dos procedimentos `add1`, que incrementa o seu argumento de uma unidade, e `sub1`, que decrementa o seu argumento de uma unidade.

```
(define (soma a b)
  (if (zero? a)
      b
      (add1 (soma (sub1 a) b))))

(define (soma a b)
  (if (zero? a)
      b
      (soma (sub1 a) (add1 b))))
```

Usando o modelo da substituição, ilustre o processo gerado por cada procedimento ao avaliar `(soma 4 5)`. Estes processos são iterativos ou recursivos? Porquê?

Exercício 4.2

Para cada um dos seguintes procedimentos, que geram processos recursivos, defina novas versões dos procedimentos que gerem processos iterativos.

- ```
(define (produto x y)
 (if (zero? x)
 0
 (+ y (produto (- x 1) y))))
```
- ```
(define (div x y)
  (if (< x y)
      0
      (+ 1 (div (- x y) y))))
```

Exercício 4.3

Utilizando os procedimentos `zero?` e `sub1`, defina o procedimento `par?`, que verifica se um número inteiro superior ou igual a zero é par ou não.

- Usando um processo recursivo
- Usando um processo iterativo

Exercício 4.4

Com base em somas e subtrações, defina o procedimento `produto`, que calcula o produto entre dois inteiros superiores ou iguais a zero x e y .

- Usando um processo recursivo
- Usando um processo iterativo

Exercício 4.5

Defina um procedimento que calcula uma potência inteira de x usando um processo iterativo. Note que $x^n = x * x^{n-1}$ e $x^0 = 1$.

Modifique o procedimento anterior para que passe também a conseguir calcular potências em que o expoente é negativo. Note que $x^{-n} = \frac{1}{x^n}$.

Exercício 4.6

Escreva um procedimento que utilize um processo iterativo para calcular o valor de $\text{sen}(x)$ utilizando a expansão em série:

$$\text{sen}(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Assuma que já existem os procedimentos `fact` e `pot` que calculam o factorial e a potência, respectivamente.

O seu procedimento deve receber, para além do valor em radianos para o qual quer calcular o seno, o número de termos que devem ser considerados.

Exercício 4.7

Defina um procedimento para calcular aproximações de π , usando a fórmula

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \dots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \dots}$$

O procedimento deve receber o número de termos a utilizar para calcular a aproximação.

1. Usando um processo recursivo
2. Usando um processo iterativo

5 Série Nº5

Exercício 5.1

O número de combinações de m objectos n a n pode ser calculado pela seguinte função:

$$Comb(m, n) = \begin{cases} 1 & \text{se } n = 0, \\ 1 & \text{se } n = m, \\ Comb(m-1, n) + Comb(m-1, n-1) & \text{se } m > n, m > 0 \text{ e } n > 0. \end{cases}$$

1. Escreva um procedimento que calcula o número de combinações de m objectos n a n . Use a estrutura de blocos para garantir que o seu procedimento recebe sempre os argumentos correctos: inteiros superiores ou iguais a zero e $m \geq n$.
2. Sabendo que existem 49 números possíveis para o totoloto e que cada chave tem 6 números diferentes, calcule o número de chaves existentes.
3. Sabendo que cada aposta custa 40\$00, quanto dinheiro teria que gastar para ter a certeza que ganhava um primeiro prémio?

Exercício 5.2

(Livro — 1.15) O seno de um ângulo (especificado em radianos) pode ser calculado usando a aproximação $\sin x \approx x$ se x for suficientemente pequeno, e a identidade trigonométrica

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

para reduzir o valor do argumento de \sin . (Para este exercício, vamos considerar que um ângulo é “suficientemente pequeno” se a sua magnitude não for maior que 0.1 radianos.)

Estas ideias estão incorporadas nos procedimentos seguintes:

```
(define (cube x)
  (* x x x))

(define (sine angle)
  (define (p x)
    (- (* 3 x)
       (* 4 (cube x))))
  (if (<= (abs angle) 0.1)
      angle
      (p (sine (/ angle 3)))))
```

1. Quantas vezes é que o procedimento `p` é aplicado quando avaliamos `(sine 12.5)`?
2. Qual é a ordem de crescimento em espaço e número de passos (em função de a) usados pelo processo gerado pelo procedimento `sine` quando avaliamos `(sine a)`?

Exercício 5.3

Considere definido o procedimento `sum`:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
          (sum term (next a) next b))))
```

Diga o que fazem as seguintes chamadas a esse procedimento:

1. `(sum (lambda (x) x) 4 add1 500)`
2. `(sum (lambda (x) (sqrt x)) 5 (lambda (x) (+ x 5)) 500)`
3. `(sum (lambda (x) (sum (lambda (x) x) 1 add1 x)) 1 add1 5)`

Exercício 5.4

(Livro — 1.30) O procedimento `sum` apresentado acima gera recursão linear. No entanto, pode ser escrito de forma a gerar um processo iterativo. Mostre como é que isso poderia ser feito preenchendo as expressões que faltam na definição que se segue:

```
(define (sum term a next b)
  (define (iter a result)
    (if <??>
        <??>
        (iter <??> <??>)))
  (iter <??> <??>))
```

Exercício 5.5

Com base no procedimento `sum`, escreva um procedimento para calcular o valor de $\text{sen}(x)$ utilizando a expansão em série:

$$\text{sen}(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Assuma que já existem os procedimentos `fact` e `pot` que calculam o factorial e a potência, respectivamente.

O seu procedimento deve receber, para além de x , o número n de termos que devem ser considerados.

Exercício 5.6

(Livro — 1.31)

1. O procedimento `sum` é apenas o mais simples de um vasto número de abstrações semelhantes, que podem ser capturadas como procedimentos de ordem superior. Escreva um procedimento análogo chamado `product`, que retorna o produto dos valores de uma função para pontos pertencentes a um intervalo. Mostre como definir o `factorial` em termos do `product`. Use também o `product` para calcular aproximações de π usando a fórmula

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \dots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \dots}$$

2. Se o seu procedimento `product` gerar um processo recursivo, escreva um que gere um processo iterativo. Se gerar um processo iterativo, escreva um que gere um processo recursivo.

Exercício 5.7

(Livro — 1.32)

1. Mostre que `sum` e `product` são ambos casos especiais de uma noção ainda mais geral chamada `accumulate`, que combina uma coleção de termos, usando uma função de acumulação geral:

```
(accumulate combiner null-value term a next b)
```

`Accumulate` recebe como argumentos o mesmo `term` e as mesmas especificações do intervalo `a` e `b`, bem como um procedimento `combiner` (de 2 argumentos) que especifica como é que o termo corrente deve ser combinado com a acumulação dos termos precedentes e um `null-value`, que especifica qual o valor a usar quando os termos acabam. Escreva o procedimento `accumulate` e mostre como é que `sum` e `product` podem ser definidos como simples chamadas a `accumulate`.

2. Se o seu procedimento `accumulate` gerar um processo recursivo, escreva um que gere um processo iterativo. Se gerar um processo iterativo, escreva um que gere um processo recursivo.

Exercício 5.8

(Livro — 1.34) Suponha que definimos o procedimento

```
(define (f g)
  (g 2))
```

Assim, temos:

```
(f quadrado)
4
```

```
(f (lambda (z) (* z (+ z 1))))
6
```

O que acontece se (perversamente) pedirmos ao interpretador para avaliar `(f f)`? Explique.

6 Série Nº6

Exercício 6.1

Considere a seguinte expressão matemática: $3x! + 4(x!)^3$

1. Escreva um procedimento `calc-expr` que calcule o seu valor.
2. Usando a estrutura de blocos, garanta que o seu procedimento recebe sempre um argumento correcto ($x \geq 0$).
3. Comente as afirmações seguintes:
 - (a) Neste caso, não havia necessidade de utilizar a estrutura de blocos.
 - (b) Neste caso, convém utilizar a forma especial `let`.
 - (c) Neste caso, não devo definir o procedimento `cubo`.
 - (d) O procedimento `cubo`, se for definido, deve ser definido dentro do procedimento `calc-expr`.

Exercício 6.2

Defina um procedimento `curry` que recebe dois argumentos. O primeiro argumento deve ser um procedimento de dois argumentos. O procedimento `curry` deve retornar um procedimento de modo a que a expressão $((\text{curry } p \ x) \ y)$ produza o mesmo resultado que $(p \ x \ y)$. Ou seja, o valor retornado é um procedimento de um argumento que corresponde ao procedimento recebido com o seu primeiro argumento fixo, sendo igual ao segundo argumento recebido pelo procedimento `curry`.

Exercício 6.3

Utilizando o procedimento `curry` do exercício 6.2, defina os seguintes procedimentos:

1. O procedimento `add1`.
2. O procedimento `triplo` que calcula o triplo de um número.
3. O procedimento `potencia-de-2` que recebe um número n e calcula o valor de 2^n (sabendo que existe definido em Scheme o procedimento `expt` que calcula a potência de um número a um determinado expoente).

Exercício 6.4

Utilizando a definição do procedimento `curry` do exercício 6.2, explique que resultados produzem as seguintes expressões:

1. `(curry curry +)`
2. `(curry curry curry)`

Exercício 6.5

Considere a seguinte expressão:

```
((((p 1) 2) 3 4) 5)
```

1. O que pode concluir acerca de p e de cada uma das sub-expressões da expressão acima?
2. Defina p de forma a que a avaliação da expressão acima produza o resultado de somar todos os números, ou seja, 15.

Exercício 6.6

(Livro — 1.41) Defina um procedimento que recebe como argumento um procedimento de um argumento e retorna um procedimento que aplica duas vezes o procedimento original.

Por exemplo, se `add1` for um procedimento que adiciona 1 ao seu argumento, então `(double add1)` deverá ser um procedimento que adiciona dois:

```
((double add1) 5)
7
(((double double) add1) 5)
9
```

Qual é o valor retornado por `((double (double double)) add1) 5)`? Porquê?

Exercício 6.7

(Livro — 1.42) Sejam f e g duas funções de um argumento. A composição f depois de g é definida como sendo a função $x \mapsto f(g(x))$. Defina um procedimento `compose` que implementa a composição. Por exemplo, se `inc` for um procedimento que adiciona 1 ao seu argumento,

```
((compose square inc) 6)
49
```

Exercício 6.8

(Livro — 1.43) Se f for uma função numérica e n um inteiro positivo, então podemos formar a n -ésima repetição da aplicação de f , que é definida como a função cujo valor em x é $f(f(\dots(f(x))\dots))$.

Por exemplo, se f for a função $x \mapsto x + 1$, então a n -ésima repetição da aplicação de f é a função $x \mapsto x + n$.

Se f for a operação de elevar um número ao quadrado, então a n -ésima repetição da aplicação de f é a função que eleva o seu argumento a 2^n .

Escreva um procedimento chamado `repeated`, que recebe como argumentos um procedimento que calcula f e um inteiro positivo n e retorna um procedimento que calcula a n -ésima repetição da aplicação de f . O seu procedimento deverá poder ser usado da seguinte forma:

```
((repeated square 2) 5)
625
```

Sugestão: Pode ser conveniente usar o `compose` do exercício anterior.

Exercício 6.9

(Livro — 1.44) A ideia de alisar uma função é um conceito importante em processamento de sinal. Se f é uma função e dx é um número pequeno, então a versão alisada de f é a função cujo valor no ponto x é a média de $f(x - dx)$, $f(x)$ e $f(x + dx)$.

Escreva um procedimento `smooth` que recebe como argumento um procedimento que calcula f e retorna um procedimento que calcula f alisada.

Algumas vezes, pode ser útil alisar repetidamente uma função (isto é, alisar a função alisada e assim sucessivamente) para obter a função alisada n -vezes. Mostre como é que poderia gerar a função alisada n -vezes de qualquer função usando `smooth` e `repeated` do exercício anterior.

Exercício 6.10

Suponha que tem definido o procedimento `comb`, que recebe dois inteiros maiores ou iguais a zero, m e n , e calcula o número de combinações de m elementos n a n .

Sabendo que existem 49 números possíveis para o totoloto e que cada chave tem 6 números diferentes, escreva um procedimento que escreve quanto dinheiro tem que ser gasto para ter a certeza de ganhar o primeiro prémio do totoloto, se as apostas custassem 40\$00, 50\$00 ou 60\$00.

7 Série Nº7

Exercício 7.1

Diga qual o resultado de avaliar cada uma das seguintes expressões. Se alguma delas der origem a um erro, explique porquê.

```
(cons 2 3)
```

```
(car (cons 2 3))
```

```
(caddr (cons 2 3))
```

```
(cdr (cons "ola" "bom dia"))
```

```
(cdr (cons 2 3))
```

```
(cons (integer? (sqrt 4)) (integer? 2.0))
```

```
(pair? (cons 2 3))
```

```
(cons (cons 1 2) (cons 3 4))
```

Exercício 7.2

Represente os seguintes pares usando a notação de caixas e ponteiros. Diga como consegue construir esses pares usando o procedimento `cons`:

1. (1 . 2)
2. (1 . (2 . (3 . (4 . 5))))
3. (1 . (2 . 3))

Exercício 7.3

Considere as seguintes definições para o procedimento `make-rat`, que, dados dois inteiros, retorna o racional em que o primeiro é o numerador e o segundo é o denominador:

```
(define (make-rat n d)
  (cons n d))
```

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

Em relação à primeira definição, a segunda tem a vantagem de reduzir o numerador e o denominador aos números mais pequenos possível.

(Livro — 2.1) Defina uma versão melhor de `make-rat` que considere argumentos positivos e negativos. `make-rat` deve normalizar o sinal, de forma a que, se o número racional for positivo, tanto o numerador como o denominador são positivos; e se o número racional for negativo, só o numerador é que é negativo.

Exercício 7.4

(Livro — 2.2) Considere o problema de representar segmentos de recta num plano. Cada segmento é representado por um par de pontos: um ponto inicial e um ponto final.

Defina um construtor `make-segment` e os selectores `start-segment` e `end-segment` que definem a representação dos segmentos em termos de pontos.

Adicionalmente, um ponto pode ser representado como um par de números: a coordenada x e a coordenada y .

Especifique o construtor `make-point` e os selectores `x-point` e `y-point` que definem esta representação.

Usando os seus selectores e construtores, defina um procedimento `midpoint-segment` que recebe um segmento de recta como argumento e retorna o seu ponto médio (o ponto cujas coordenadas são a média das coordenadas dos pontos que definem o segmento).

Exercício 7.5

(Livro — 2.3) Implemente uma representação de rectângulos num plano. (Pode ser útil usar os resultados do exercício anterior.)

Com base nos seus construtores e selectores, crie procedimentos que calculem o perímetro e a área de um dado rectângulo.

Implemente uma representação diferente para os rectângulos.

Consegue conceber o seu sistema com as barreiras de abstracção adequadas, de forma a que os procedimentos que calculam a área e o perímetro funcionem com qualquer das representações?

Exercício 7.6

Com base nas respostas aos exercícios anteriores, escreva um procedimento `dentro-rectangulo`, que recebe um rectângulo e um ponto e retorna `#t` se o ponto estiver dentro do rectângulo (incluindo a fronteira) e `#f` se estiver fora do rectângulo.

8 Série Nº8

Exercício 8.1

Diga qual o resultado de avaliar cada uma das seguintes expressões. Se alguma delas der origem a um erro, explique porquê.

```
(list 1 2 3)
```

```
(list (cons 1 3) 4)
```

```
(cdr (list 2 3))
```

```
(car (cdr (list 2 3)))
```

```
(cdr (cdr (list 2 3)))
```

```
()
```

```
(list ())
```

```
(list)
```

```
(list? (cons 2 3))
```

```
(list? (cons 2 ()))
```

```
(list? (list 2 3))
```

```
(pair? (list 2 3 4))
```

```
(null? (list 1 2))
```

```
(null? ())
```

```
(null? (list ()))
```

Exercício 8.2

Represente as seguintes listas usando a notação de caixas e ponteiros:

1. (1)

2. (1 2)

3. (1 (2 (3 (4 5))))

4. (1 (2 . 3) 4)

5. (((2 (6 (7 . 8) 3)) 1))

6. (1 (((2))))

Exercício 8.3

Defina os seguintes procedimentos que operam sobre listas. Os seus procedimentos devem dar erro (usando o `error`) quando isso se justificar. Quando for possível, escreva dois procedimentos, um que gera um processo recursivo e outro que gera um processo iterativo.

1. O procedimento `primeiro-par` que recebe uma lista e retorna um par com os dois primeiros elementos da lista.
2. O procedimento `maior-elemento` que recebe uma lista de inteiros e retorna o maior elemento dessa lista.
3. O procedimento `soma-elementos` que recebe uma lista e retorna a soma de todos os elementos dessa lista.
4. O procedimento `imprime-lista-de-pares` que recebe uma lista de pares e imprime os pares, um por linha. O seu procedimento deve assinalar quando é que chega ao fim da lista. Por exemplo,

```
(imprime-lista-de-pares (list (cons "Luisa" 12345678)
                              (cons "Jorge" 23456789)
                              (cons "Maria" 34567890)
                              (cons "Rui" 45678901)))
```

Deverá imprimir

```
Luisa -> 12345678
Jorge  -> 23456789
Maria  -> 34567890
Rui    -> 45678901
Fim da lista
```

Exercício 8.4

(Livro — 2.17) Defina um procedimento `last-pair`, que retorna a lista que contém apenas o último elemento de uma dada lista não vazia:

```
>(last-pair (list 23 72 149 34))
(34)
>(last-pair ())
cdr: expects argument of type <pair>; given ()
>(last-pair (list ()))
(())
```

Exercício 8.5

(Livro — 2.18) Defina um procedimento `reverse`, que recebe como argumento uma lista e retorna uma lista com os mesmos elementos, mas pela ordem inversa:

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

Exercício 8.6

Defina um procedimento `map`, que recebe como argumentos um procedimento de um argumento e uma lista, e retorna a lista dos resultados produzidos aplicando o procedimento a cada elemento da lista.

```
(map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)
```

Exercício 8.7

(Livro — 2.21) O procedimento `square-list` recebe como argumento uma lista de números e retorna uma lista com os quadrados desses números.

```
(square-list (list 1 2 3 4))
(1 4 9 16)
```

Seguem-se duas definições diferentes para o procedimento `square-list`. Complete ambas as definições, preenchendo as expressões que faltam:

```
(define (square-list items)
  (if (null? items)
      ()
      (cons <??> <??>)))
```

```
(define (square-list items)
  (map <??> <??>))
```

Exercício 8.8

(Livro — 2.22) O Luís tenta re-escrever o primeiro procedimento `square-list` do exercício anterior de modo a que ele passe a gerar um processo iterativo:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons ((lambda (x) (* x x)) (car items))
                    answer))))
  (iter items ()))
```

Infelizmente, definir o procedimento `square-list` desta maneira produz a lista de resposta pela ordem inversa à desejada. Porquê?

O Luís tenta então corrigir este erro trocando os argumentos do `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    ((lambda (x) (* x x)) (car items))))))
  (iter items ()))
```

Isto também não funciona. Explique porquê.

Exercício 8.9

(Livro — 2.23) O procedimento `for-each` é semelhante ao `map`. Recebe como argumentos um procedimento e uma lista de elementos. No entanto, em vez de formar uma lista com os resultados, `for-each` apenas aplica o procedimento a cada um dos elementos de cada vez, da esquerda para a direita. Os valores retornados pela aplicação do procedimento aos elementos não são usados — `for-each` é usado com procedimentos que executam uma acção, tal como imprimir. Por exemplo:

```
(for-each (lambda (x) (newline) (display x))
          (list 57 321 28))
57
321
28
```

O valor retornado pela chamada a `for-each` (não ilustrado acima) pode ser qualquer coisa, como verdadeiro. Apresente uma implementação para o procedimento `for-each`.

Exercício 8.10

Implemente o procedimento `imprime-lista-de-pares` do exercício 8.3 usando o procedimento `for-each`. Lembre-se que o procedimento recebe uma lista de pares e imprime os pares, um por linha, e deve assinalar quando é que chega ao fim da lista. Por exemplo,

```
(imprime-lista-de-pares (list (cons "Luisa" 12345678)
                              (cons "Jorge" 23456789)
                              (cons "Maria" 34567890)
                              (cons "Rui" 45678901)))
```

Deverá imprimir

```
Luisa -> 12345678
Jorge -> 23456789
Maria -> 34567890
Rui -> 45678901
Fim da lista
```

Exercício 8.11

(Livro — 2.24) Suponha que avaliemos a expressão `(list 1 (list 2 (list 3 4)))`. Mostre o resultado impresso pelo interpretador, a estrutura de caixas e ponteiros correspondente.

Exercício 8.12

(Livro — 2.25) Apresente combinações de `cars` e `cdrs` que seleccionem o 7 de cada uma das listas seguintes:

```
(1 3 (5 7) 9)
```

```
((7))
```

```
(1 (2 (3 (4 (5 (6 7)))))
```

Exercício 8.13

(Livro — 2.26) Suponha que definimos `x` e `y` como sendo duas listas:

```
(define x (list 1 2 3))
```

```
(define y (list 4 5 6))
```

Qual é o resultado impresso pelo interpretador como resposta a cada uma das seguintes expressões?

```
(append x y)
```

```
(cons x y)
```

```
(list x y)
```

9 Série Nº9

Exercício 9.1

Considere que foi definido o tipo árvore binária. Para este tipo, estão definidas as operações:

- `constroi-arvore` que recebe a raiz, a árvore esquerda e a árvore direita e constrói a árvore correspondente.
- `arvore-raiz` que recebe uma árvore binária e retorna a sua raiz.
- `arvore-esquerda` que recebe uma árvore binária e retorna a sua árvore esquerda.
- `arvore-direita` que recebe uma árvore binária e retorna a sua árvore direita.
- `arvore-vazia?` que recebe um objecto e retorna verdadeiro se ele corresponder a uma árvore vazia e falso caso contrário.

Com base nas operações descritas, escreva os seguintes procedimentos para percorrer árvores binárias:

1. `percorre-inorder` recebe uma árvore binária e retorna uma lista com todas as suas folhas, percorrendo primeiro a árvore esquerda, depois a raiz e depois a árvore direita da árvore inicial.
2. `percorre-preorder` recebe uma árvore binária e retorna uma lista com todas as suas folhas, percorrendo primeiro a raiz, depois a árvore esquerda e depois a árvore direita da árvore inicial.
3. `percorre-posorder` recebe uma árvore binária e retorna uma lista com todas as suas folhas, percorrendo primeiro a árvore esquerda, depois a árvore direita e depois a raiz da árvore inicial.

Exercício 9.2

Uma árvore binária de procura é uma árvore binária em que todos os elementos que estão na sua árvore esquerda são menores que a raiz e todos os elementos que estão na sua árvore direita são maiores que a raiz.

Com base nas operações definidas no exercício anterior, escreva os seguintes procedimentos:

1. `insere-elemento` que recebe um elemento e uma árvore binária de procura e o insere na árvore.
2. `ordena-lista` que recebe uma lista de elementos e retorna uma nova lista com os elementos ordenados.

Exercício 9.3

Escreva um procedimento `filtra` que recebe um predicado e uma lista e retorna uma lista que contém apenas os elementos da lista inicial que satisfazem o predicado. Por exemplo:

```
(filtra even? (list 1 2 3 4 5))
(2 4)
```

```
(filtra even? (list 1 3 5 7))
()
```

Exercício 9.4

Escreva um procedimento `todos?` que recebe um predicado e uma lista e retorna verdadeiro se todos os elementos da lista satisfizerem o predicado e falso caso contrário. Por exemplo:

```
(todos? even? (list 1 2 3 4 5))
#f
```

```
(todos? even? (list 2 4 6))
#t
```

Exercício 9.5

Escreva um procedimento `algum?` que recebe um predicado e uma lista e retorna verdadeiro se algum dos elementos da lista satisfizer o predicado e falso caso contrário. Por exemplo:

```
(algum? odd? (list 1 2 3 4 5))
#t
```

```
(algum? odd? (list 2 4 6))
#f
```

Exercício 9.6

Escreva um procedimento `substitui` que recebe dois elementos e uma lista e retorna uma outra lista que resulta de substituir todas as ocorrências do primeiro elemento pelo segundo na lista inicial. Por exemplo:

```
(substitui 2 3 (list 1 2 3 2 5))
(1 3 3 3 5)
```

```
(substitui 2 3 (list 1 3 5 7))
(1 3 5 7)
```

Exercício 9.7

Escreva um procedimento `fold-right` que recebe um procedimento de dois argumentos, o valor inicial de um acumulador e uma lista e retorna o resultado de aplicar o procedimento ao elemento inicial e ao resultado de aplicar o procedimento a todos os elementos que estão à sua direita. Quando a lista for vazia, este procedimento deve retornar o valor inicial. Por exemplo:

```
(fold-right + 0 (list 1 2 3 4))
10
```

```
(fold-right + 0 ())
0
```

Exercício 9.8

Com base no procedimento `fold-right` escreva os seguintes procedimentos:

1. `multiplica-lista` que recebe uma lista e retorna o produto de todos os seus elementos.

2. `maximo-lista` que recebe uma lista e retorna o maior dos seus elementos.
3. `inverte-lista` que recebe uma lista e retorna outra lista com os elementos da lista inicial pela ordem inversa.
4. `junta-listas` que recebe duas listas e retorna outra lista que resulta de juntar as duas.

Exercício 9.9

Uma forma de compactar listas de números é, dada uma lista de números (possivelmente repetidos), transformá-la numa lista em que ocorrências consecutivas de um mesmo número são substituídas por um par, em que o primeiro elemento é o número de vezes que o número aparece repetido e o segundo elemento é o número.

Escreva o procedimento `run-length-encode` que compacta listas de inteiros. Por exemplo,

```
(run-length-encode '(1 1 1 1 1 1 1 2 3 3 3 3 4 4 4 4 1 3 3 3 3))
((7 . 1) 2 (4 . 3) (4 . 4) 1 (4 . 3))
```

```
(run-length-encode '(1 2 1 2 3 3 3 3 4 4 4 4 1 1 3 3 3 3 3))
(1 2 1 2 (4 . 3) (4 . 4) (2 . 1) (5 . 3))
```

Repare que as sequências de apenas um elemento não são substituídas.

Depois de ter uma lista compactada, pode ser necessário saber qual era a lista original. Escreva o procedimento `run-length-decode` que, dada uma lista de inteiros compactada, retorna a lista original. Por exemplo,

```
(run-length-decode '((7 . 1) 2 (4 . 3) (4 . 4) 1 (4 . 3)))
(1 1 1 1 1 1 1 2 3 3 3 3 4 4 4 4 1 3 3 3 3)
```

```
(run-length-decode '(1 2 1 2 (4 . 3) (4 . 4) (2 . 1) (5 . 3)))
(1 2 1 2 3 3 3 3 4 4 4 4 1 1 3 3 3 3 3)
```

Exercício 9.10

(Livro — 2.27) Modifique o procedimento `reverse` (do Livro — 2.18) para produzir um procedimento `deep-reverse` que recebe uma lista como argumento e retorna a lista com os seus elementos invertidos e com todas as suas sublistas também invertidas. Por exemplo,

```
(define x (list (list 1 2) (list 3 4)))
```

```
x
((1 2) (3 4))
```

```
(reverse x)
((3 4) (1 2))
```

```
(deep-reverse x)
((4 3) (2 1))
```

Lembre-se que o procedimento `reverse` recebe como argumento uma lista e retorna uma lista com os mesmos elementos, mas pela ordem inversa:

```
(define (reverse l)
  (define (r-aux l res)
    (if (null? l)
        res
        (r-aux (cdr l) (cons (car l) res))))
  (r-aux l ()))
```

Exercício 9.11

(Livro — 2.28) Escreva um procedimento `fringe` que recebe como argumento uma árvore (representada como uma lista de listas) e retorna uma lista cujos elementos são todas as folhas da árvore da esquerda para a direita. Por exemplo,

```
(define x (list (list 1 2) (list 3 4)))

(fringe x)
(1 2 3 4)

(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

Exercício 9.12

(Livro — 2.30) Defina o procedimento `square-tree` análogo ao `square-list` (do Livro — 2.21). O procedimento `square-tree` deve-se comportar da seguinte forma:

```
(square-tree
 (list 1
      (list 2 (list 3 4) 5)
      (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Deve definir este procedimento directamente (isto é, sem usar procedimentos de ordem superior) e também usando o procedimento `map`.

Exercício 9.13

(Livro — 2.31) Abstraia a sua resposta ao exercício anterior para produzir um procedimento `tree-map`, com a propriedade que `square-tree` poderia ser definido como:

```
(define (square-tree tree)
  (tree-map square tree))
```

Exercício 9.14

(Livro — 2.32) Podemos representar um conjunto como uma lista de elementos distintos, e podemos representar o conjunto de todos os subconjuntos de um conjunto como uma lista de listas. Por exemplo, se o conjunto é $(1\ 2\ 3)$, então o conjunto de todos os seus subconjuntos é $((\) (3) (2) (2\ 3) (1) (1\ 3) (1\ 2) (1\ 2\ 3))$. Complete a seguinte definição de um procedimento que gera o conjunto dos subconjuntos de um conjunto e dê uma explicação clara de porque é que ele funciona.

```
(define (subsets s)
  (if (null? s)
      (list ())
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

10 Série Nº10

Exercício 10.1

(Livro — exemplo das páginas 143-4) Considere que foram feitas as definições:

```
(define a 1)
(define b 2)
```

Diga qual o valor de cada uma das seguintes expressões:

```
(list a b)
(list 'a 'b)
(list 'a b)
(first '(a b c))
(rest '(a b c))
```

Exercício 10.2

(Livro — exemplo da página 144) Defina o procedimento `memq`, que recebe um símbolo e uma lista e retorna falso se o símbolo não estiver contido na lista (isto é, não for `eq?` a nenhum dos elementos da lista) e a sublista que começa com a primeira ocorrência do símbolo na lista caso contrário. Por exemplo,

```
(memq 'apple '(pear banana prune))
#f

(memq 'apple '(x (apple sauce) y apple pear))
(apple pear)
```

Exercício 10.3

(Livro — 2.53) O que é que o interpretador de Scheme imprime como resposta à avaliação de cada uma das seguintes expressões:

```
(list 'a 'b 'c)
(list (list 'george))
(cdr '((x1 x2) (y1 y2)))
(cadr '((x1 x2) (y1 y2)))
(pair? (car '(a short list)))
(memq 'red '((red shoes) (blue socks)))
(memq 'red '(red shoes blue socks))
```

Exercício 10.4

(Livro — 2.54) Duas listas são `equal?` se contiverem elementos iguais e estes estiverem pela mesma ordem. Por exemplo,

```
(equal? '(this is a list) '(this is a list))
```

é verdade, mas

```
(equal? '(this is a list) '(this (is a) list))
```

é falso. Para sermos mais precisos, podemos definir `equal?` recursivamente em termos da igualdade básica entre símbolos `eq?`, dizendo que `a` e `b` são `equal?` se forem ambos símbolos e forem `eq?` ou forem ambas listas em que `(first a)` é `equal?` a `(first b)` e `(rest a)` é `equal?` a `(rest b)`. Usando esta ideia, implemente `equal?` como um procedimento.

Exercício 10.5

(Livro — 2.55) O resultado de avaliar a expressão

```
(first 'abracadabra)
```

é `quote`. Explique porquê.

Exercício 10.6

Escreva um procedimento `count-if` que recebe um predicado e uma lista e retorna o número de elementos da lista que satisfazem o predicado. Por exemplo:

```
(count-if even? '(1 2 3 2 5))
2
```

```
(count-if (curry eq? 'a) '(a b (a) c))
1
```

Exercício 10.7

Escreva um procedimento `position` que recebe um objecto e uma lista e retorna a primeira posição em que ocorre um elemento na lista `eq?` ao objecto dado. Se não existir nenhum objecto nessas condições na lista o procedimento deve devolver `#f`. A posição do primeiro elemento da lista é a 0 (zero). Por exemplo:

```
(position 'a '(a b c a b))
0
```

```
(position 'a '(b c a b))
2
```

```
(position 'd '(b c a b))
#f
```

Exercício 10.8

Escreva um procedimento `complement` que recebe um predicado de um argumento e devolve a negação desse predicado. Por exemplo:

```
((complement even?) 2)
#f
```

```
((complement even?) 3)
#t
```

Exercício 10.9

Usando os procedimentos `filtra`, `curry` e `complement` escreva um procedimento `remove` que recebe em objecto e uma lista e devolve a lista que resulta de eliminar todos os elementos `eq?` ao objecto dado da lista dada. Por exemplo:

```
(remove 'd '(a b c))  
(a b c)
```

```
(remove 'a '(a b c a b a))  
(b c b)
```

Exercício 10.10

Escreva um procedimento `adjoin` que recebe um elemento e uma lista e adiciona o elemento à lista apenas se ele ainda não existir na lista (de acordo com o `eq?`). Por exemplo:

```
(adjoin 'a '(a b c))  
(a b c)
```

```
(adjoin 'd '(a b c))  
(d a b c)
```

Exercício 10.11

Escreva um procedimento `remove-duplicates` que recebe uma lista e devolve uma lista sem elementos repetidos (de acordo com o `eq?`). Por exemplo:

```
(remove-duplicates '(a b c))  
(a b c)
```

```
(remove-duplicates '(a b a a c a))  
(b c a)
```

Exercício 10.12

Escreva um procedimento `merge` que recebe duas listas de números por ordem crescente e deve devolver uma lista com todos os números das duas listas ordenados por ordem crescente. Por exemplo:

```
(merge '(1 4 7 10) '(2 4 5 12))  
(1 2 4 4 5 7 10 12)
```

Exercício 10.13

Escreva uma versão do procedimento `substitui` utilizando o procedimento `map`.

Exercício 10.14

Escreva um procedimento `substitui-arvore` que corresponde ao procedimento `substitui` para árvores, ou seja, que substitua todas as ocorrências de um determinado objecto por outro na árvore.

Exercício 10.15

Escreva um procedimento `profundidade` que recebe uma árvore e devolve um número que indica qual é o nível mais profundo de listas dentro dessa lista, aumentando de um sempre que se entra para dentro de uma lista. Se a lista não contém listas devolve 0. Por exemplo:

```
(profundidade '(1 2 3))  
0
```

```
(profundidade '(1 2 (3 ((4) (5 (6)))))  
4
```

Exercício 10.16

Escreva um procedimento `full-position` que recebe um objecto e uma árvore e devolve uma lista com as posições sucessivas a considerar para atingir o objecto (que pode estar em qualquer nível da árvore). Considere que a primeira posição de cada (sub-)lista é a zero. Utilize o procedimento `eq?` para comparar elementos. Por exemplo:

```
(full-position 3 '(1 2 3 4 5))  
(2)  
(full-position 'h '(a (b) c (d ((e f) g (h)))))  
(3 1 2 0)
```

11 Série Nº11

Exercício 11.1

Diga o que é impresso pelo interpretador de Scheme ao avaliar cada uma das seguintes expressões:

```
(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)
```

```
(set! c 78)
```

Exercício 11.2

(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 11.3

(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 11.4

Defina um procedimento `vector-map` que corresponde ao procedimento `map`, mas para vetores: recebe um procedimento de um argumento e um vetor e retorna um vetor com os resultados produzidos ao aplicar o procedimento a cada elemento do vetor 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 11.5

Defina um procedimento `vector-map-into!`, semelhante ao procedimento `vector-map` do exercício anterior, mas que altera destrutivamente o vetor recebido substituindo os elementos do vetor 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 11.6

Defina um procedimento `vector-reverse!` que recebe um vetor e inverte esse vetor, 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 11.7

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 11.8

O procedimento `vector-search` do exercício anterior 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 11.9

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?

12 Série Nº12

Exercício 12.1

Desenhe o diagrama dos ambientes criados 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 12.2

Desenhe o diagrama dos 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 12.3

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

<pre>(let ((x 1) (y 2)) (let ((x 4) (z (+ x 4))) (set! y (+ x z)) (display (+ x y z))) (+ x y))</pre>	<pre>(let ((x 1) (y 2)) (let ((x 4) (z (+ x 4))) (define y (+ x z)) (display (+ x y z))) (+ x y))</pre>
---	---

Exercício 12.4

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 dos ambientes criados pelo código acima.

Exercício 12.5

Introduzir a forma especial `set!` na nossa linguagem obriga-nos a pensar no significado de igualdade e mudança. Dê exemplos de procedimentos simples que sejam:

1. Um procedimento referencialmente transparente
2. Um procedimento referencialmente opaco (não transparente)