



UNIVERSIDADE TÉCNICA DE LISBOA  
INSTITUTO SUPERIOR TÉCNICO

Resumos e Soluções para os  
Exercícios para as Aulas Práticas da Disciplina de  
Fundamentos da Programação

Ana Cardoso Cachopo

Ano Lectivo 2006/2007



## Prefácio

Este documento contém os exercícios que eu vou resolver nas aulas práticas em 2006/07.

É baseado nos exercícios e soluções de 1998/99 e 1999/00.

Os resumos da matéria são baseados no livro “PROGRAMAÇÃO EM SCHEME: Introdução à Programação com Múltiplos Paradigmas”, de João Pavão Martins e Maria dos Remédios Cravo, 2004, IST Press.

Quando aparece a referência a um exercício do Sussman, diz respeito a “Structure and Interpretation of Computer Programs”, de Harold Abelson e Gerald Jay Sussman e Julie Sussman, Second Edition, 1996, MIT Press.

## **Conteúdo**

<b>1</b>	<b>Introdução ao Scheme; Notação Prefixa</b>	<b>1</b>
<b>2</b>	<b>Avaliação de Expressões; Procedimentos Simples e Compostos</b>	<b>5</b>
<b>3</b>	<b>Recursão; Estrutura de Blocos</b>	<b>17</b>
<b>4</b>	<b>Processos Iterativos; Processos Recursivos</b>	<b>25</b>
<b>5</b>	<b>Procedimentos de Ordem Superior</b>	<b>33</b>
<b>6</b>	<b>Procedimentos de Ordem Superior (cont.)</b>	<b>39</b>
<b>7</b>	<b>O Tipo Lista</b>	<b>45</b>
<b>8</b>	<b>Funcionais Sobre Listas</b>	<b>59</b>
<b>9</b>	<b>Símbolos, Listas e Árvores Binárias</b>	<b>71</b>
<b>10</b>	<b>Programação Imperativa</b>	<b>85</b>
<b>11</b>	<b>Avaliação Baseada em Ambientes</b>	<b>95</b>
<b>12</b>	<b>O Tipo Vector</b>	<b>101</b>
<b>13</b>	<b>Estruturas Mutáveis: Filas, Pilhas e Estruturas Circulares</b>	<b>107</b>

# 1 Introdução ao Scheme; Notação Prefixa

## Sumário:

- Apresentação
- Instruções de instalação do Scheme
- Introdução ao Scheme

## Resumo:

- Apresentação
  - O meu nome, quem está 1ª opção, quem tem computador e internet em casa, o que estão a achar do IST, praxes, mentorado...
  - Na faculdade é preciso trabalhar: 2 TPC, 4 minitests, 2 testes e 1 projecto significam elementos de avaliação quase todas as semanas.
  - É suposto virem para as aulas de problemas com a matéria teórica sabida, indo às aulas e lendo o livro em casa.
  - Não deixar atrasar a matéria.
  - Vão aos horários de dúvidas!
  - Objectivo da cadeira: ensinar a programar **bem** em qualquer linguagem. Queremos escrever bons programas e não apenas programas que funcionem. Os bons programas devem ser fáceis de perceber, fáceis de manter, fáceis de alterar, modulares, com nomes bem escolhidos...
  - Ferramenta usada: linguagem Scheme.
    - \* Porque não é conhecida, por isso não há maus hábitos adquiridos.
    - \* Porque tem uma sintaxe simples, basicamente (`<operador> <args>`), usando uma notação prefixa.
- Instruções de instalação do Scheme — ver enunciado do primeiro TPC. O trabalho deve vir numa fonte `monospace`.
- Introdução ao Scheme
  - Um *programa* é um algoritmo escrito numa linguagem de programação.
  - Um *processo computacional* é um ente imaterial que existe dentro de um computador durante a execução de um programa, e cuja evolução ao longo do tempo é ditada pelo programa.
  - Um *programa em Scheme* é uma sequência de formas.  
`<programa em Scheme> ::= <forma>*`
  - Uma *forma* é uma frase em Scheme.  
`<forma> ::= <definição> | <expressão>`
  - Uma *expressão* é uma entidade computacional que tem um valor.  
`<expressão> ::= <constante> | <combinação> | <nome>`

- \* O valor de uma *constante* é a própria constante. Ex: números inteiros 1, 456, números racionais  $2/3$ ,  $345/456$ , números reais 2.1, 4.567,  $8.9e+10$ , valores lógicos #t, #f (não existe t nem nil), cadeias de caracteres "ola", ...
  - \* Uma *combinação* corresponde à aplicação de uma operação a uma sequência de zero ou mais operandos.  
<combinação> ::= (<operador> <expressão>\*)
  - \* Os *nomes* são definidos pelo programador.
  - \* Um *predicado* corresponde a uma operação que produz resultados do tipo lógico.
  - \* Uma *condição* é uma expressão cujo valor é do tipo lógico.
- O *ciclo lê-avalia-escreve* corresponde à utilização interactiva do interpretador de uma linguagem, em que o interpretador lê uma forma, avalia essa forma, e escreve os resultados dessa avaliação. Pode ser tão longo quanto se desejar.

**Exercício 1.1**

Traduza as seguintes expressões matemáticas para a notação prefixa, tendo o cuidado de garantir que as operações são efectuadas exactamente pela mesma ordem que seriam efectuadas em matemática:

1.  $2 + 3$
2.  $5 + 4 + 3 + 2 + 1$
3.  $5^3 + 4^2$
4.  $(5 + 4)^3$
5.  $2 - 3 - 4$
6.  $2/3/4/5$
7.  $\sqrt{3 + 6}$
8.  $5 + 4 \cdot 3/2 + 1$
9.  $(5 + 4) \cdot 3/(2 + 1)$
10.  $(5 + 4 \cdot 3)/(2 + 1)$
11.  $5 - 4 \cdot (3/2 + 1)$
12.  $1 + 2 \cdot 3/4 - 5 \cdot 6/7 + 8$
13. (Sussman — 1.2)  $\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}$

**Resposta:**

1. (+ 2 3)
2. (+ 5 4 3 2 1)
3. (+ (\* 5 5 5) (\* 4 4))

Uma das vantagens da notação prefixa é que podemos ter operadores com número variável de argumentos, pois o seu âmbito nunca é ambíguo.

Na notação matemática temos que conhecer os operadores matemáticos e as suas precedências para sabermos a ordem das operações. Na notação prefixa usada em Scheme não é ambíguo qual das operações é efectuada em cada ponto.

4. (\* (+ 5 4)  
   (+ 5 4)  
   (+ 5 4))

Usar impressão elegante facilita a leitura, pois é mais fácil saber quais são os parâmetros de cada operador, e facilita a escrita, pois é mais fácil identificar quais os parêntesis a fechar.

5. (- 2 3 4)  
   ou  
   (- (- 2 3) 4)  
   e não  
   (- 2 (- 3 4))

6. (/ 2 3 4 5)

ou

(/ (/ (/ 2 3) 4) 5)

e não

(/ 2 (/ 3 (/ 4 5)))

7. (sqrt (+ 3 6))

8. (+ 5

(/ (\* 4 3)

2)

1)

9. (/ (\* (+ 5 4) 3)

(+ 2 1))

10. (/ (+ 5 (\* 4 3))

(+ 2 1))

11. (- 5

(\* 4

(+ (/ 3 2)

1)))

12. (+ (- (+ 1

(/ (\* 2 3)

4))

(/ (\* 5 6)

7)))

8)

13. (/ (+ 5

4

(- 2

(- 3

(+ 6 (/ 4 5))))))

(\* 3

(- 6 2)

(- 2 7)))

O que dá, depois de avaliado,

-37/150

Se algum dos números fosse um real, o resultado seria

-0.246666666666667

## 2 Avaliação de Expressões; Procedimentos Simples e Compostos

### Sumário:

- Avaliação de expressões
- Nomes
- Formas especiais
- Procedimentos simples e compostos
- Expressões condicionais

### Resumo:

- Avaliação de expressões
  1. Se a expressão é uma constante, o seu valor é a própria constante.
  2. Se a expressão é um nome, o seu valor é o objecto computacional associado ao nome (que pode ser um procedimento).
  3. Se a expressão é uma forma especial, o seu valor é calculado pelas regras de avaliação dessa forma especial.
  4. Se a expressão é uma combinação, aplica-se o operador da combinação aos operandos, o que é feito do seguinte modo:
    - (a) Avaliam-se as subexpressões na combinação (por qualquer ordem).
    - (b) Associam-se os parâmetros formais do procedimento correspondente à primeira subexpressão (o operador) com os valores das restantes subexpressões (os parâmetros concretos), pela ordem respectiva.
    - (c) No ambiente definido pela associação entre os parâmetros formais e os parâmetros concretos, avalia-se o corpo do procedimento correspondente à primeira subexpressão.
- Em Scheme podemos associar *nomes* a valores.  
`<operação de nomeação> ::= (define <nome> <expressão>)`  
 Um *nome* é qualquer sequência de caracteres que comece com uma letra. Um *ambiente* é uma associação entre nomes e objectos computacionais. O valor de uma expressão depende do ambiente em que é avaliada. O valor de um nome é o valor associado a esse nome no ambiente em consideração.
- As *formas especiais* têm regras de avaliação especiais. Ex: `define`, `and`, `or`, ...
- Procedimentos simples e compostos
  - Procedimentos *simples* são os que estão incluídos no interpretador do Scheme.  
Ex: `+`, `sin`, `or`, ...
  - Procedimentos *compostos* são os definidos pelo programador e são usados na abstracção procedimental: interessa-nos *o que* o procedimento faz e não *como* o faz. Podemos definir procedimentos em Scheme:

- \* Através de expressões lambda, quando não lhes queremos dar nome. São os chamados procedimentos anónimos. Ex: (lambda (x) (+ x 3))  
 <expressão lambda> ::= (lambda (<parâmetros formais>) <corpo>)  
 <parâmetros formais> ::= <nome>\*  
 <corpo> ::= <definição>\* <expressão>+
  - \* Dando-lhes um nome, quando os podemos querer usar mais do que uma vez. Ex: (define (quadrado x) (\* x x))  
 <operação de nomeação> ::= (define <nome> <expressão>) |  
 (define (<nome> <parâmetros formais>) <corpo>)
- As *expressões condicionais* são formas especiais  
 <expressão condicional> ::= (cond <cláusula>+) |  
 (cond <cláusula>+ <última cláusula>) |  
 (if <condição> <expressão1> <expressão2>) |  
 (if <condição> <expressão>)  
 <cláusula> ::= (<condição> <expressão>\*)  
 <última cláusula> ::= (else <expressão>+)

**Exercício 2.1**

Diga qual é o resultado de avaliar cada uma das seguintes *constant*es em Scheme. Se alguma delas não for uma constante ou der origem a um erro, explique porquê.

10

-5

10.

3/4

2/6

#f

#T

``ola''

ola

**Resposta:**

```
> 10
10
```

```
> -5
-5
```

```
> 10.
10.0
```

```
> 3/4
3/4
```

```
> 2/6
1/3
```

```
> #f
#f
```

```
> #T
#t
```

```
> "ola"
"ola"
```

```
> ola
. reference to undefined identifier: ola
```

**Notas:**

Os racionais são representados na forma canónica, isto é, o numerador e o denominador são

primos entre si.

ola não é uma constante, é um nome cujo valor não foi definido.

### Exercício 2.2

Diga qual é o resultado de avaliar cada uma das seguintes *combinações* em Scheme.

Se alguma delas não for uma combinação ou der origem a um erro, explique porquê.

(+)

(+ 5)

(+ 2 3 4 5 6)

(+ 2.3 4)

(-)

(- 5)

(- 2 3 4/8)

(\*)

(\* 5)

/

(/)

(/ 5)

(/ 2 3)

(/ 2.0 3)

(/ 2 3 4 5 6)

(not #t)

(+ 3 (> 3 4))

(and #f (<= 2 3) (> 4 3))

(or (not #t) (not (> 3 4)))

(or (= 5.0 5) (> 2 3) (= 324 (\* 56 7)))

(even? 2)

```
(odd? 3.0)

(>= 6 5 4 3 2)

(max 1 2 3.0 4)

(min 1 2/3 4.0)

(abs -5.3)

(+ (sqrt 9) (expt 2 3))

(quotient 1 2)

(remainder 1 2)

(truncate 3.7)

(sqrt 9)

(expt 2 3)
```

**Resposta:**

```
> (+)
0

> (+ 5)
5

> (+ 2 3 4 5 6)
20

> (+ 2.3 4)
6.3

> (-)
. -: expects at least 1 argument, given 0

> (- 5)
-5

> (- 2 3 4/8)
-1 1/2

> (*)
1

> (* 5)
5

> /
#<primitive:/>
```

## 10 2 AVALIAÇÃO DE EXPRESSÕES; PROCEDIMENTOS SIMPLES E COMPOSTOS

```
> (/)
. /: expects at least 1 argument, given 0

> (/ 5)
1/5

> (/ 2 3)
2/3

> (/ 2.0 3)
0.6666666666666666

> (/ 2 3 4 5 6)
1/180

> (not #t)
#f

> (+ 3 (> 3 4))
. +: expects type <number> as 2nd argument, given: #f; other arguments were: 3

> (and #f (<= 2 3) (> 4 3))
#f

> (or (= 5.0 5) (> 2 3) (= 324 (* 56 7)))
#f

> (or #t)
#t

> (even? 2)
#t

> (odd? 3.0)
#t

> (>= 6 5 4 3 2)
#t

> (max 1 2 3.0 4)
4.0

> (min 1 2/3 4.0)
0.6666666666666666

> (abs -5.3)
5.3

> (+ (sqrt 9) (expt 2 3))
11

> (quotient 1 2)
0

> (remainder 1 2)
```

```

1
> (truncate 3.7)
3.0

> (sqrt 9)
3

> (expt 2 3)
8

```

Nota: / não é uma combinação, é o nome de uma primitiva.

### Exercício 2.3

(Sussman — 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))

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

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

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

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

(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(and (<= a b) (even? a) (odd? b))

(max (quotient a 2) (remainder b 2))

```

**Resposta:**

```

> 10
10

> (+ 5 3 4)
12

> (- 9 1)
8

> (/ 6 2)
3

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

> (+ (* (+ (* 2 3) 4) 5) 6)
56

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

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

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

> (define a 3)

> (define b (+ a 1))

> (+ a b (* a b))
19

> (= a b)
#f

>(and (<= a b) (even? a) (odd? b))
#f

>(max (quotient a 2) (remainder b 2))
1

```

**Exercício 2.4**

Em baixo é apresentada uma sequência de expressões. Diga qual é o resultado impresso pelo interpretador de Scheme quando é avaliada cada uma delas. Assuma que a sequência é avaliada pela ordem apresentada.

```

((lambda (x) (remainder x 3)) 7)

((lambda (x y z) (max x y z)) 2 3 4)

((lambda (x y z) (max x y z)) 2 3 4 5)

```

```
(lambda (x y z) (max x y z))

((lambda (x) (if (odd? x) "impar" "par")) 7)

((lambda (x) (if (even? x) "par" "impar")) 0)

((lambda (x)
  (cond ((< x 0) "negativo")
        ((= x 0) "zero")
        (else "positivo"))))
234)
```

**Resposta:**

```
> ((lambda (x) (remainder x 3)) 7)
1

> ((lambda (x y z) (max x y z)) 2 3 4)
4

> ((lambda (x y z) (max x y z)) 2 3 4 5)
.#<procedure>: expects 3 arguments, given 4: 2 3 4 5

> (lambda (x y z) (max x y z))
.#<procedure>

> ((lambda (x) (if (odd? x) "impar" "par")) 7)
"impar"

> ((lambda (x) (if (even? x) "par" "impar")) 0)
"par"

> ((lambda (x)
  (cond ((< x 0) "negativo")
        ((= x 0) "zero")
        (else "positivo"))))
234)
"positivo"
```

**Exercício 2.5**

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

**Resposta:**

```
(define (perimetro-circunferencia raio)
  (* 2
     pi
     raio))

(define pi 3.1415927)
```

**Exercício 2.6**

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

**Resposta:**

```
(define (area-circunferencia raio)
  (* pi
    (quadrado raio)))

(define (quadrado x)
  (* x x))

(define pi 3.1415927)
```

**Exercício 2.7**

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

**Resposta:**

```
(define (volume-esfera raio)
  (* (/ 4 3)
    pi
    (cubo raio)))

(define (cubo x)
  (* x x x))

(define pi 3.1415927)
```

**Exercício 2.8**

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

**Resposta:**

```
(define (volume-casca-esferica raio1 raio2)
  (- (volume raio2) (volume raio1)))
```

**Exercício 2.9**

Defina um procedimento anônimo em Scheme que recebe dois números como argumentos e devolve o maior deles. Deve fazê-lo em 3 versões: uma usando o `if`, outra usando o `cond` e outra usando o `max`.

**Resposta:**

Utilizando o `if`:

```
(lambda (x y)
  (if (> x y)
      x
      y))
```

Utilizando o cond:

```
(lambda (x y)
  (cond ((> x y) x)
        (else y)))
```

Utilizando o procedimento max:

```
(lambda (x y)
  (max x y))
```

### Exercício 2.10

Defina o procedimento `max3` que recebe três números como argumentos e devolve o maior deles. Deve fazê-lo em 3 versões: uma usando o `if`, outra usando o `cond` e outra usando o `max`.

**Resposta:**

Utilizando o cond:

```
(define (max3 x y z)
  (cond ((and (>= x y) (>= x z)) x)
        ((and (>= y x) (>= y z)) y)
        (else z)))
```

Nota: tem que ser `>=` senão `(max3 5 5 2)` dava 2 e não 5.

Utilizando ifs:

```
(define (max3 x y z)
  (if (> x y)
      (if (> x z) ; aqui, -----+-----+-----> só vale a pena
          x ; y x comparar x e z
          z) ; porque y não é o maior
      (if (> y z) ; aqui, -----+-----+-----> só vale a pena
          y ; x y comparar y e z
          z))) ; porque x não é o maior
```

Utilizando o procedimento max:

```
(define (max3 x y z)
  (max x y z))
```



### 3 Recursão; Estrutura de Blocos

#### Sumário:

- Recursão
- Estrutura de blocos
- Nomes locais

#### Resumo:

- Nomes locais  
 $\langle \text{expressão let} \rangle ::= (\text{let } (\langle \text{nome e expressão} \rangle+) \langle \text{corpo} \rangle)$   
 $\langle \text{nome e expressão} \rangle ::= (\langle \text{nome} \rangle \langle \text{expressão} \rangle)$   
 Para quando é necessário usar lets encadeados, existe o  $\text{let}^*$   
 $\langle \text{expressão let}^* \rangle ::= (\text{let } (\langle \text{nome e expressão} \rangle+) \langle \text{corpo} \rangle)$
- Estrutura de blocos
  - Um bloco corresponde a uma sequência de instruções e deve corresponder a um subproblema que o procedimento original tem que resolver. Este aspecto permite modularizar o problema.
  - O algoritmo usado por um bloco está escondido do resto do programa. Isto permite controlar a complexidade do problema.
  - Toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Isto permite a protecção da informação definida em cada bloco da utilização não autorizada por parte de outros blocos.
- O *ambiente global* contém todos os nomes que foram fornecidos directamente ao interpretador do Scheme.
- Um *ambiente local* corresponde a uma associação entre nomes e objectos computacionais, a qual é realizada durante o processo de avaliação de uma forma. Informalmente, um ambiente local “desaparece” no momento em que termina a avaliação da forma que deu origem à sua criação.
- Um nome está *ligado* num dado ambiente se existe um objecto computacional associado ao nome nesse ambiente.
- Um *nome local* apenas tem significado dentro do corpo de uma expressão lambda.
- Um nome que aparece no corpo de uma expressão lambda e que não é um nome local dise *não local*.
- O *domínio* de um nome é a gama de formas nas quais o nome é conhecido, isto é, o conjunto das formas onde o nome pode ser utilizado.
- O tipo de domínio utilizado no Scheme é chamado *domínio estático*: o domínio de um nome é definido em termos da estrutura do programa (o encadeamento dos seus blocos) e não é influenciado pelo modo como a execução do programa é feita.

- Uma definição *recursiva* é uma definição que é feita à custa de si própria.

**Exercício 3.1**

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, estaria errado definir o procedimento `cubo`.
  - (d) O procedimento `cubo`, se for definido, deve ser definido dentro do procedimento `calc-expr`.

**Resposta:**

```
1. (define (fact x)
    (if (zero? x)
        1
        (* x (fact (sub1 x)))))

(define (calc-expr x)
  (let ((x-fact (fact x)))
    (+ (* 3 x-fact)
       (* 4 (expt x-fact 3)))))
```

```
2. (define (calc-expr x)

  (define (calc-expr-aux x)
    (let ((x-fact (fact x)))
      (+ (* 3 x-fact)
         (* 4 (expt x-fact 3)))))

  (if (>= x 0)
      (calc-expr-aux x)
      (error "calc-expr: o arg deve ser um inteiro positivo.")))
```

Nota: a forma correcta de proceder no caso de os argumentos não serem o que se está à espera é lançar um erro, mas isso ainda não foi dado. Basicamente, usa-se o procedimento `error`, que lança um erro e mostra a string que lhe foi passada como argumento.

3. (a) Neste caso, não havia necessidade de utilizar a estrutura de blocos: É verdade que não havia necessidade de utilizar a estrutura de blocos, pois o teste se o argumento é válido só seria feito uma vez, mesmo que fosse feito no procedimento exterior. Faria sentido usar a estrutura de blocos se o procedimento fosse recursivo, para evitar a repetição do teste a cada iteração, o que deveria ter sido feito para definir o `factorial`.
- (b) Neste caso, convém utilizar a forma especial `let`: É verdade que convém utilizar a forma especial `let`, pois assim podemos guardar o valor de  $x!$ , que de outra forma teria que ser calculado duas vezes.
- (c) Neste caso, não deve ser definido o procedimento `cubo`: Esta afirmação não é totalmente verdadeira, pois aquilo que o procedimento `cubo` faz é uma operação bem definida e que pode ser útil noutros procedimentos. Assim, pode fazer sentido definir este procedimento (embora nós tenhamos usado o `expt`).

- (d) O procedimento `cubo`, se for definido, deve ser definido dentro do procedimento `calc-expr`: Tal como já dissémos, o procedimento `cubo` pode ser útil noutros procedimentos. Por isso, se for definido, deve ser definido fora do procedimento `calc-expr`.

### Exercício 3.2

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 50 números e 9 estrelas possíveis para o Euromilhões e que cada chave tem 5 números e duas estrelas diferentes, calcule o número de chaves existentes.
3. Sabendo que cada aposta custa 2 Euro, quanto dinheiro teria que gastar para ter a certeza que ganhava um primeiro prémio?
4. Pretende-se agora saber quanto seria necessário gastar se as apostas do Euromilhões tivessem diferentes valores. Escreva um procedimento que mostre quanto é que é necessário gastar se cada aposta custar 2Euro, 2.5Euro, ou 3Euro.

### Resposta:

1. 

```
(define (comb m n)

  (define (comb-aux m n)
    (cond ((= n 0) 1)
          ((= m n) 1)
          (else (+ (comb-aux (- m 1) n)
                   (comb-aux (- m 1) (- n 1))))))

  (if (and (>= m n) (> m 0) (> n 0))
      (comb-aux m n)
      (error "comb: args devem ser inteiros positivos e 1o > 2o")))
```
2. Para calcular o número de chaves existentes para o Euromilhões, avaliar 

```
(* (comb 50 5) (comb 9 2))
```

 (pode demorar algum tempo). O resultado deverá ser 76275360.
3. Para saber quanto dinheiro teríamos que gastar para ter a certeza de ganhar o primeiro prémio, basta multiplicar este valor por 2 Euro, que é quanto custa cada aposta: 

```
(* 76275360 2)
```

, o que dá 152550720 Euro. Ou seja, teríamos de gastar mais de 152 milhões de Euro.
4. 

```
(define (precosEuromilhoes)
  (let ((numApostas (comb 49 6)))
    (display "Se as apostas custarem 2Euro, tem que gastar ")
    (display (* 2 numApostas))))
```

```
(display "Euro")
(newline)
(display "Se as apostas custarem 2.5Euro, tem que gastar ")
(display (* 2.5 numApostas))
(display "Euro")
(newline)
(display "Se as apostas custarem 3Euro, tem que gastar ")
(display (* 3 numApostas))
(display "Euro")
(newline))
```

(precosEuromilhoes)

É essencial usar o `let`, para não ter computações repetidas.

### Exercício 3.3

Considere o cálculo de potências inteiras de um número.

1. Defina um procedimento que calcula uma potência inteira de  $x$ , sabendo que  $x^n = x * x^{n-1}$  e  $x^0 = 1$ . Nota: não pode usar o procedimento `expt` do Scheme.
2. Modifique o procedimento anterior para que passe também a conseguir calcular potências em que o expoente é negativo, sabendo que  $x^{-n} = \frac{1}{x^n}$ .
3. Modifique o procedimento anterior para que passe a tratar correctamente os casos  $0^n$  com  $n$  negativo (divisão por zero), e  $0^0$  (indeterminação).

### Resposta:

1. 

```
(define (potencia x n)
  (if (= n 0)
      1
      (* x (potencia x (- n 1)))))
```

2. Para tratar os expoentes negativos, temos que testar, no procedimento exterior, qual o valor do expoente, e chamar o procedimento auxiliar da maneira adequada:

```
(define (potencia x n)

  (define (pot x n)
    (if (= n 0)
        1
        (* x (pot x (- n 1)))))

  (if (< n 0)
      (/ 1 (pot x (abs n)))
      (pot x n)))
```

3. Para tratar estes casos, temos que fazer os testes correspondentes no procedimento exterior e chamar o procedimento auxiliar da maneira adequada:

```
(define (potencia x n)

  (define (pot x n)
    (if (= n 0)
        1
```

```

(* x (pot x (- n 1))))

(cond ((< n 0)
      (if (= x 0)
          (error "potencia: divisao por zero")
          (/ 1 (pot x (abs n)))))
      ((and (= n 0) (= x 0))
       (error "potencia: indeterminacao"))
      (else
       (pot x n))))

```

### Exercício 3.4

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?`.

### Resposta:

1. 

```
(define (soma x y)
  (if (zero? x)
      y
      (add1 (soma (sub1 x) y))))
```
2. 

```
(define (igual? x y)
  (cond ((zero? x) (zero? y))
        ((zero? y) #f)
        (else (igual? (sub1 x) (sub1 y)))))
```
3. 

```
(define (menor? x y)
  (cond ((zero? y) #f)
        ((zero? x) #t)
        (else (menor? (sub1 x) (sub1 y)))))
```

4. 

```
(define (diferenca x y)
  (if (zero? y)
      x
      (sub1 (diferenca x (sub1 y)))))
```
- ou
- ```
(define (diferenca x y)
  (if (zero? y)
      x
      (diferenca (sub1 x) (sub1 y))))
```
5. 

```
(define (produto x y)
  (if (zero? x)
      0
      (soma y
            (produto (sub1 x) y))))
```
6. 

```
(define (divisao-inteira x y)
  (if (menor? x y)
      0
      (add1 (divisao-inteira (diferenca x y) y))))
```

### Exercício 3.5

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?`.

### Resposta:

1. 

```
(define (ganha? n)
  (or (<= n 3)
      (perde? (- n 1))
      (perde? (- n 2))
      (perde? (- n 3))))
```
2. 

```
(define (perde? n)
  (not (ganha? n)))
```

```
3. (define (nim n)
  (cond ((<= n 3) n)
        ((perde? (- n 3)) 3)
        ((perde? (- n 2)) 2)
        ((perde? (- n 1)) 1)
        (else 1)))
```

## 4 Processos Iterativos; Processos Recursivos

### Sumário:

- Processos iterativos
- Processos recursivos

### Resumo:

- Uma definição *recursiva* é uma definição que é feita à custa de si própria. Um procedimento recursivo tanto pode originar um processo recursivo como um processo iterativo.
- Num *processo recursivo* existe uma fase de expansão correspondente à construção de uma cadeia de operações adiadas, seguida por uma fase de contracção correspondente à execução dessas operações. A um processo recursivo que cresce linearmente com um valor dá-se o nome de processo *recursivo linear*.
- Um *processo iterativo* é caracterizado por um certo número de variáveis, chamadas variáveis de estado, juntamente com uma regra que especifica como actualizá-las, e não cresce nem se contrai. As variáveis fornecem uma descrição completa do estado da computação em cada momento. A um processo iterativo cujo número de operações cresce linearmente com um valor dá-se o nome de processo *iterativo linear*.
- Na *recursão em árvore* existem múltiplas fases de expansão e de contracção, originadas pela múltipla recursão do procedimento que a origina (o procedimento refere-se várias vezes a si próprio). Geralmente, a complexidade destes processos cresce exponencialmente com um valor.
- `(display <string>)`
- `(string-append <string>*)`
- `(newline)`
- A *sequenciação* permite especificar que uma dada sequência de expressões deve ser avaliada pela ordem em que aparece.  
`<operação de sequenciação> ::= (begin <expressão>+)`
- A *ordem de crescimento* de um processo é uma medida grosseira dos recursos computacionais (tempo e espaço de memória) consumidos pelo processo em função do grau de dificuldade do problema.

**Exercício 4.1**

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  $x^{-n} = \frac{1}{x^n}$ .

**Resposta:**

Tal como já vimos, se usássemos um processo recursivo, ficaria:

```
(define (potencia x n)
  (if (zero? n)
      1
      (* x (potencia x (sub1 n)))))
```

Alternativamente, podemos usar um processo iterativo, usando um procedimento auxiliar com mais um argumento que vai funcionar como acumulador:

```
(define (potencia x n)

  (define (pot x n acc)
    (if (zero? n)
        acc
        (pot x (sub1 n) (* acc x))))

  (pot x n 1))
```

Para tratar os expoentes negativos, temos que testar, no procedimento exterior, qual o valor do expoente, e chamar o procedimento auxiliar da maneira adequada:

```
(define (potencia x n)

  (define (pot x n acc)
    (if (zero? n)
        acc
        (pot x (sub1 n) (* acc x))))

  (if (< n 0)
      (/ 1 (pot x (abs n) 1))
      (pot x n 1)))
```

**Exercício 4.2**

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

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

**Resposta:**

Usando um processo recursivo, fica:

```
(define (produto x y)
  (if (zero? y)
      0
      (+ x (produto x (sub1 y)))))
```

Usando um processo iterativo, fica:

```
(define (produto x y)
  (define (prod x y acc)
    (if (zero? y)
        acc
        (prod x (sub1 y) (+ acc x))))
  (prod x y 0))
```

### Exercício 4.3

(Adaptado de Sussman — 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?

**Resposta:**

No primeiro caso, temos um processo recursivo, onde podemos facilmente identificar a fase de expansão e a fase de contracção:

```
(soma 4 5)
(add1 (soma 3 5))
(add1 (add1 (soma 2 5)))
(add1 (add1 (add1 (soma 1 5))))
(add1 (add1 (add1 (add1 (soma 0 5)))))
(add1 (add1 (add1 (add1 5))))
(add1 (add1 (add1 6)))
(add1 (add1 7))
(add1 8)
9
```

No segundo caso, temos um processo iterativo, onde não existe expansão nem contracção:

```
(soma 4 5)
(soma 3 6)
(soma 2 7)
(soma 1 8)
(soma 0 9)
9
```

**Exercício 4.4**

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

1. 

```
(define (produto x y)
  (if (zero? x)
      0
      (+ y (produto (- x 1) y))))
```
2. 

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

**Resposta:**

A ideia fundamental da passagem de um processo recursivo para um processo iterativo (convém notar que o procedimento auxiliar continua a ser recursivo porque é definido à custa de si próprio) é ir acumulando os resultados parciais num novo argumento, inicializado da primeira vez que se chama o procedimento auxiliar.

1. 

```
(define (produto x y)
  (define (prod-iter x y res)
    (if (zero? x)
        res
        (prod-iter (- x 1) y (+ y res))))
  (prod-iter x y 0))
```

ou, como o `y` não é modificado no `prod-iter`

```
(define (produto x y)
  (define (prod-iter x res)
    (if (zero? x)
        res
        (prod-iter (- x 1) (+ y res))))
  (prod-iter x 0))
```

Mas, uma vez que o número de argumentos é reduzido, é melhor prática de programação continuar a passar todos os argumentos.

2. 

```
(define (div x y)
  (define (div-iter x y res)
    (if (< x y)
        res
        (div-iter (- x y) y (+ res 1))))
  (div-iter x y 0))
```

**Exercício 4.5**

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.

1. Usando um processo recursivo

## 2. Usando um processo iterativo

**Resposta:**

## 1. Usando um processo recursivo, fica:

```
(define (par? n)
  (if (zero? n)
      #t
      (not (par? (sub1 n)))))
```

## 2. Usando um processo iterativo, fica:

```
(define (par? n)
  (define (par-iter? n res)
    (if (zero? n)
        res
        (par-iter? (sub1 n) (not res))))
  (par-iter? n #t))
```

**Exercício 4.6**

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

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

## 1. Usando um processo recursivo

## 2. Usando um processo iterativo

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

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

**Resposta:**

Definição dos procedimentos auxiliares

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (sub1 n)))))
```

```
(define (pot x n)
  (if (= n 0)
      1
      (* x (pot x (sub1 n)))))
```

Convém reparar que

$$\text{sen}(x) = \sum_{i=1}^n \frac{x^{(2i-1)}}{(2i-1)!} \cdot (-1)^{i+1}$$

## 1. Usando um processo recursivo

```
(define (seno x n)
  (if (< n 1)
      0
      (+ (seno x (- n 1))
          (let ((k (- (* n 2) 1)))
            (* (pot -1 (+ n 1))
                (/ (pot x k)
                    (fact k))))))))
```

ou, se preferirmos,

```
(define (seno x n)
  (define (sum k sinal)
    (if (> k (* 2 n))
        0
        (+ (* sinal (/ (pot x k) (fact k)))
            (sum (+ k 2) (* -1 sinal)))))
  (sum 1 1))
```

Nota: existe a convenção que, se um argumento não for alterado por um procedimento nem pelos procedimentos definidos dentro dele, não deve ser passado como argumento para os procedimentos interiores (porque nunca muda).

## 2. Usando um processo iterativo

```
(define (seno x n)
  (define (sum k sinal acc)
    (if (> k (* 2 n))
        acc
        (sum (+ k 2)
              (* -1 sinal)
              (+ (* sinal (/ (pot x k) (fact k)))
                  acc))))
  (sum 1 1 0))
```

**Exercício 4.7**

Defina um procedimento para calcular aproximações de  $\pi$ , 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

**Resposta:**

Para calcular as aproximações de  $\pi$ , podemos separar a fórmula anterior em duas, e considerar que

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

No entanto, esta solução tem o problema de que tanto o numerador como o denominador atingem valores muito altos e fazem overflow.

Uma alternativa é considerar que a série, desde o termo 1 até ao termo  $n$  vai progredindo da seguinte forma: nos termos de índice par, o numerador é o índice mais dois e o denominador o índice mais um; nos termos de índice ímpar, o numerador é o índice mais um e o denominador o índice mais dois. Assim, ficamos com os procedimentos seguintes:

### 1. Usando um processo recursivo

```
(define (pi n)
  (define (term k)
    (if (even? k)
        (/ (+ k 2) (+ k 1))
        (/ (+ k 1) (+ k 2))))
  (define (calc-pi/4 n)
    (if (= n 0)
        1
        (* (term n)
           (calc-pi/4 (- n 1)))))
  (* 4 (calc-pi/4 n)))
```

### 2. Usando um processo iterativo

```
(define (pi n)
  (define (term k)
    (if (even? k)
        (/ (+ k 2) (+ k 1))
        (/ (+ k 1) (+ k 2))))
  (define (calc-pi/4 n res)
    (if (= n 0)
        res
        (calc-pi/4 (- n 1) (* res (term n)))))
  (* 4 (calc-pi/4 n 1)))
```



## 5 Procedimentos de Ordem Superior

### Sumário:

- Procedimentos de ordem superior — procedimentos como parâmetros de procedimentos.

### Resumo:

- Um objecto computacional é um *cidadão de primeira classe* se:
  1. pode ser nomeado;
  2. pode ser utilizado como argumento de procedimentos;
  3. pode ser devolvido por procedimentos;
  4. pode ser utilizado como componente de estruturas de informação.
- Um procedimento que recebe outros procedimentos como parâmetros ou cujo valor é um procedimento é chamado um *procedimento de ordem superior* ou, alternativamente, um funcional.
- Os procedimentos podem ser usados como parâmetros para outros procedimentos.
- Os procedimentos podem ser o resultado de outros procedimentos.

**Exercício 5.1**

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)`

**Resposta:**

1. Soma os números entre 4 e 500.
2. Soma as raízes quadradas dos múltiplos de 5 entre 5 e 500.
3. Para cada um dos números entre 1 e 5, soma os números entre 1 e esse número:  
 $(1 + (1 + 2) + (1 + 2 + 3) + (1 + 2 + 3 + 4) + (1 + 2 + 3 + 4 + 5))$

**Exercício 5.2**

(Sussman — 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 <??> <??>))
```

**Resposta:**

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

**Exercício 5.3**

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.

**Resposta:**

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(define (seno-com-sum x n)
  (if (< n 1)
      (display "Tem que considerar pelo menos um termo.")
      (sum (lambda (n)
             (* (expt -1 (- n 1))
                (/ (expt x (- (* 2 n) 1))
                   (fact (- (* 2 n) 1)))))
            1
            add1
            n)))
```

Outra possibilidade para definir o procedimento `seno-com-sum` seria definir um procedimento auxiliar chamado, por exemplo, `term`. Tanto no caso de usarmos a `lambda` como no caso do procedimento auxiliar, temos que nos lembrar que o `sum` recebe um procedimento de um argumento. Por isso, mesmo que quiséssemos, por questão de clareza incluir mais argumentos, neste caso isso não seria possível. Com o procedimento auxiliar, `seno-com-sum` ficaria:

```
(define (seno-com-sum x n)
  (define (term n)
    (* (expt -1 (- n 1))
       (/ (expt x (- (* 2 n) 1))
          (fact (- (* 2 n) 1)))))
  (if (< n 1)
      0
      (sum term 1 add1 n)))
```

**Exercício 5.4**

(Sussman — 1.29) A Regra de Simpson é um método para fazer integração numérica. Usando a Regra de Simpson, o integral de uma função  $f$  entre  $a$  e  $b$  pode ser aproximado por

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n]$$

onde  $h = (b - a)/n$ , para algum inteiro par  $n$ , e  $y_k = f(a + kh)$ . (Aumentar o  $n$  aumenta a precisão da aproximação.) Defina um procedimento que recebe como argumentos  $f$ ,  $a$ ,  $b$  e  $n$  e retorna o valor do integral, calculado usando a Regra de Simpson. Use o seu procedimento para integrar o procedimento `cubo` entre 0 e 1 (com  $n = 100$  e  $n = 1000$ ), e compare os resultados com os do procedimento `integral` apresentado na página 60 do livro.

Nota: Deve usar o procedimento `sum`, definido na página 58 do livro, como

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

**Resposta:**

### Exercício 5.5

(Sussman — 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  $\pi$  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.

**Resposta:**

```
1. (define (product term a next b)
  (if (> a b)
      1
      (* (term a)
          (product term (next a) next b))))
```

```
(define (fact n)
  (product (lambda (x) x) 1 add1 n))
```

Para calcular as aproximações de  $\pi$ , podemos separar a fórmula anterior em duas, e considerar que

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

Podemos usar o `product` para calcular o valor de cada uma destas fracções. Depois, basta multiplicar o resultado por 4 para ficarmos com o valor de  $\pi$ :

```
(define (pi n)
  (define (numerator n) (* 2 n))
  (define (denominator n) (add1 (* 2 n)))
  (* 4 (* (/ (product numerator 1.0 add1 (/ n 2))
            (product denominator 1.0 add1 (/ n 2)))
        (/ (product numerator 2.0 add1 (add1 (/ n 2)))
            (product denominator 1.0 add1 (/ n 2))))))
```

No entanto, esta versão funciona mal, porque tanto o numerador como o denominador atingem valores muito altos e fazem overflow.

Mas também podemos considerar que a série, desde o termo 1 até ao termo  $n$  vai progredindo da seguinte forma: nos termos de índice par, o numerador é o índice mais dois e o denominador o índice mais um; nos termos de índice ímpar, o numerador é o índice mais um e o denominador o índice mais dois. Assim, ficamos com o procedimento seguinte:

```
(define (pi n)
  (define (term k)
    (if (even? k)
        (/ (+ k 2) (+ k 1))
        (/ (+ k 1) (+ k 2))))
  (* 4 (product term 1 add1 n)))
```

2. (define (product term a next b)
 (define (iter a result)
 (if (> a b)
 result
 (iter (next a)
 (\* (term a) result))))
 (iter a 1))

### Exercício 5.6

(Sussman — 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.

### Resposta:

1. (define (accumulate combiner null-value term a next b)

```

    (if (> a b)
        null-value
        (combiner (term a)
                  (accumulate combiner
                              null-value
                              term
                              (next a)
                              next
                              b))))

(define (sum term a next b)
  (accumulate + 0 term a next b))

(define (product term a next b)
  (accumulate * 1 term a next b))

2. (define (accumulate combiner null-value term a next b)
    (define (iter a result)
      (if (> a b)
          null-value
          (iter (next a)
                (combiner (term a) result))))
      (iter a null-value))

```

**Exercício 5.7**

(Sussman — 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.

**Resposta:**

Ao avaliar `(f f)`, temos a seguinte sequência de avaliações:

```

(f f)
(f 2)
(2 2)

```

Como 2 não é o nome de nenhum procedimento, ao avaliar a última expressão vamos obter um erro, pois não é possível aplicar o “procedimento” 2 a nenhuns argumentos.

## 6 Procedimentos de Ordem Superior (cont.)

### Sumário:

- Procedimentos de ordem superior — procedimentos retornados por procedimentos.

### Resumo:

**Exercício 6.1**

(Sussman — 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. O procedimento `compose` é um procedimento de dois argumentos (as funções a compôr) que retorna um procedimento de um argumento. Por exemplo, se `add1` for um procedimento que adiciona 1 ao seu argumento,

```
> ((compose square add1) 6)
49
```

Com base no procedimento `compose`, escreva um procedimento `triplo-fact`, que calcula o triplo do factorial do seu argumento. Por exemplo,

```
> (triplo-fact 3)
18
```

**Resposta:**

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (triplo-fact x)
  (compose (lambda (x) (* x 3)) fact x))
```

**Exercício 6.2**

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
```

**Resposta:**

```
(define (complement pred)
  (lambda (arg)
    (not (pred arg))))
```

Ou

```
(define (complement pred)
  (compose not pred))
```

**Exercício 6.3**

(Sussman — 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.

**Resposta:**

```
(define (repeated f n)
  (if (= n 1)
      f
      (compose f (repeated f (sub1 n)))))
```

**Exercício 6.4**

(Sussman — 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.

**Resposta:**

```
(define (smooth f)
  (let ((dx 0.00001))
    (lambda (x) (/ (+ (f (- x dx))
                      (f x)
                      (f (+ x dx)))
                  3))))

(define (n-fold-smooth f n)
  ((repeated smooth n) f))
```

**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.

**Resposta:**

1. Podemos concluir que  $p$  é um procedimento de um argumento que devolve um procedimento de um argumento que devolve um procedimento de dois argumentos que devolve um procedimento de um argumento.
2.

```
(define (p a)
  (lambda (b)
    (lambda (c d)
      (lambda (e)
        (+ a b c d e))))))
```

**Exercício 6.6**

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 `((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`.

**Resposta:**

```
(define (curry f arg1)
  (lambda (arg2)
    (f arg1 arg2)))
```

**Exercício 6.7**

Utilizando o procedimento `curry` do exercício 6.6, 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).

**Resposta:**

1. `(define add1 (curry + 1))`
2. `(define triplo (curry * 3))`
3. `(define potencia-de-2 (curry expt 2))`

### Exercício 6.8

Utilizando a definição do procedimento `curry` do exercício 6.6, explique que resultados produzem as seguintes expressões:

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

### Resposta:

1. A expressão `(curry curry +)` produz uma versão do procedimento `curry` especializado para o procedimento `+`, ou seja, um procedimento que, dado um número, produz um procedimento de um argumento que soma o número ao seu argumento. Por exemplo, `((curry curry +) 1)` corresponde ao procedimento `add1`.
2. A expressão `(curry curry curry)` produz uma versão “curried” do procedimento `curry`, ou seja, um procedimento equivalente a

```
(lambda (x)
  (lambda (y)
    (curry x y)))
```

Por exemplo, `(((((curry curry curry) +) 4) 5) 5)` produz o valor 9.

### Exercício 6.9

(Sussman — 1.41) Defina um procedimento `double` 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?

### Resposta:

```
(define (double proc)
  (lambda (x) (proc (proc x))))
```

```
>((double add1) 5)
7
```

```
>(((double double) add1) 5)
9
>(((double (double double)) add1) 5)
21
```

Para explicarmos estes resultados, podemos analisar os vários passos pelos quais passa a avaliação de cada uma destas expressões:

```
((double add1) 5)
((lambda (x) (add1 (add1 x))) 5)
(add1 (add1 5))
(add1 6)
7
```

Para a segunda expressão, fica:

```
((double double) add1) 5)
(((lambda (x) (double (double x))) add1) 5)
((double (double add1)) 5)
((double (lambda (x) (add1 (add1 x)))) 5)
((lambda (x2) ((lambda (x) (add1 (add1 x)))
                ((lambda (x) (add1 (add1 x))) x2))) 5)
((lambda (x) (add1 (add1 x))) (lambda (x) (add1 (add1 x)) 5))
((lambda (x) (add1 (add1 x))) (add1 (add1 5)))
((lambda (x) (add1 (add1 x))) (add1 6))
((lambda (x) (add1 (add1 x))) 7)
(add1 (add1 7))
(add1 8)
9
```

Para a última expressão, fica:

```
((double (double double)) add1) 5)
21
```

O que se passa é que o `double` aplicado a ele próprio eleva ao quadrado o número de aplicações do procedimento (em vez de o multiplicar por 2).

## 7 O Tipo Lista

### Sumário:

- Listas como Tipo Abstracto de Informação

### Resumo:

- Um *tipo de informação* é constituído por um conjunto de objectos e um conjunto de operações aplicáveis a esses objectos.
- Os *tipos de informação elementares* contêm elementos que não são decomponíveis; os *tipos de informação estruturados* contêm elementos que são compostos por várias partes.
- A *representação interna* de um objecto computacional corresponde à representação manipulada pelo interpretador de Scheme. A *representação externa* de um objecto computacional corresponde ao modo como esse objecto computacional é mostrado pelo Scheme ao mundo exterior.
- A *abstracção de dados* consiste na separação entre as partes do programa que lidam com o modo como os dados são utilizados das partes do programa que lidam com o modo como os dados são representados.
- Uma operação para combinar entidades satisfaz a *propriedade do fecho* se os resultados da combinação de entidades com essa operação puderem ser combinados através da mesma operação.
- Na *metodologia dos tipos abstractos de informação* são seguidos quatro passos sequenciais:
  1. Identificação das operações básicas. As *operações básicas* que se podem efectuar sobre os elementos do tipo dividem-se em quatro grupos:
    - (a) Os *constructores* permitem construir novos elementos do tipo.
    - (b) Os *selectores* permitem aceder aos constituintes dos elementos do tipo.
    - (c) Os *reconhecedores* identificam elementos do tipo e são de duas categorias. Por um lado, fazem a distinção entre os elementos do tipo e os elementos de que qualquer outro tipo, reconhecendo explicitamente os elementos que pertencem ao tipo. Por outro lado, identificam elementos do tipo que se individualizam dos restantes por possuírem certas propriedades particulares.
    - (d) Os *testes* efectuam comparações entre os elementos do tipo.
  2. Axiomatização das operações básicas. A *axiomatização* especifica o modo como as operações básicas se relacionam entre si.
  3. Escolha de uma representação para os elementos do tipo. A representação é escolhida em termos de outros tipos existentes.
  4. Concretização das operações básicas para a representação escolhida. O último passo na definição de um tipo de informação consiste em realizar as operações básicas definidas no primeiro passo em termos da representação definida no terceiro passo, verificando a axiomatização definida no segundo passo.

- A *encapsulação da informação* corresponde ao conceito de que o conjunto de procedimentos que representa o tipo de informação engloba toda a informação referente ao tipo.
- O *anonimato da representação* corresponde ao conceito de que o módulo correspondente aos procedimentos do tipo guarda como segredo o modo escolhido para representar os elementos do tipo.
- As *barreiras de abstracção* impedem qualquer acesso aos elementos do tipo que não seja feito através das operações básicas.
- As *estruturas dinâmicas* correspondem a estruturas de informação cujo tamanho pode variar ao longo da execução de um programa.

### O tipo *par*

- *Constructores*: cons
- *Selectores*: car, cdr
- *Reconhecedores*: pair?
- *Testes*: não identificados

### O tipo *lista*

- Uma *lista* contém uma colecção ordenada de elementos. Podemos inserir novos elementos em qualquer posição da lista (inserindo um elemento de cada vez) e podemos remover qualquer elemento da lista.
- No tipo *lista simplificada* apenas podemos manipular o primeiro elemento da lista e aceder à lista que tem todos os elementos menos o primeiro.
  - *Constructores*: nova-lista, insere
  - *Selectores*: primeiro, resto
  - *Reconhecedores*: lista?, lista-vazia?
  - *Testes*: listas=?
  - Podemos também definir o procedimento *lista*, que recebe um número arbitrário de elementos e retorna a lista que os contém. Na realidade, este procedimento é açúcar sintáctico para uma sequência de *inserir*, um para cada elemento da lista.

A definição de cada uma das operações deste tipo é apresentada de seguida, e deve ser usada pelos alunos nas aulas e no projecto.

```
(define (nova-lista) ())
```

```
(define insere cons)
```

```
(define primeiro car)
```

```
(define resto cdr)

(define lista? list?)

(define lista-vazia? null?)

(define listas=? equal?)

(define lista list)

(define junta append)
```

- No tipo *lista completa* podemos manipular os elementos em qualquer posição da lista. Parte das operações deste tipo são iguais às do tipo *lista simplificada*.
  - *Construtores*: nova-lista, insere-lista
  - *Selectores*: elem-lista, resto-lista
  - *Reconhecedores*: lista?, lista-vazia?
  - *Testes*: listas=?

Para este tipo, é necessário definir as seguintes operações:

```
(define (nova-lista) ())

(define (insere-lista el pos lst)
  (if (= pos 1)
      (cons el lst)
      (if (null? lst)
          (error "insere-lista: posicao fora da lista")
          (cons (car lst)
                (insere-lista el (- pos 1) (cdr lst))))))

(define (elem-lista pos lst)
  (if (null? lst)
      (error "elem-lista: posicao fora da lista")
      (if (= pos 1)
          (car lst)
          (elem-lista (- pos 1) (cdr lst)))))

(define (resto-lista pos lst)
  (if (null? lst)
      (error "resto-lista: posicao fora da lista")
      (if (= pos 1)
          (cdr lst)
          (cons (car lst)
                (resto-lista (- pos 1) (cdr lst))))))

(define lista? list?)
```

```
(define lista-vazia? null?)
```

```
(define listas=? equal?)
```

- Nota: o tipo *par* não foi definido nas aulas, por isso usam-se os operadores do Scheme. Só para as listas é que se usam as operações definidas para o tipo, apesar de elas também serem um tipo primitivo em Scheme.
- `()`, `cons`, `car`, `cdr`, `list`, `pair?`, `list?`, `null?`, `equal?`, `append`, `reverse`
- Não existe `atom?`, este conceito é representado como `(not (pair? ...))`.
- Outros reconhecedores para outros tipos: `integer?`, `rational?`, `real?`, `boolean?`, `string?`

**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))
```

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

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

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

```
(pair? (cons (string? 2) (string? "ola")))
```

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

**Resposta:**

```
>(cons 2 3)
(2 . 3)
```

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

```
>(cdr (cdr (cons 2 3)))
cdr: expects argument of type <pair>; given 3
```

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

```
>(cons (integer? (sqrt 4)) (integer? 2.0))
(#t . #t)
```

```
>(pair? (cons (string? 2) (string? "ola")))
#t
```

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

Nota: Por definição, os pares apenas podem ter dois elementos. O `cons` recebe sempre dois argumentos. A estrutura `((1 . 2) 3 . 4)` é uma representação alternativa para `((1 . 2) . (3 . 4))`.

**Exercício 7.2**

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

```
(1 . 2)
```

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

```
(1 . (2 . 3))
```

**Resposta:**

```
> (cons 1 2)
(1 . 2)
```

```
> (cons 1 (cons 2 (cons 3 (cons 4 5))))
(1 2 3 4 . 5)
```

```
> (cons 1 (cons 2 3))
(1 2 . 3)
```

Nota: A estrutura de pares (1 . (2 . (3 . (4 . 5)))) é equivalente a (1 2 3 4 . 5); na prática, é sempre possível substituir “ . (“ por nada, eliminando também o “)” correspondente.

**Exercício 7.3**

Represente as seguintes listas usando a notação de caixas e ponteiros. Diga como consegue construir essas listas usando os construtores do tipo *lista* definido nas aulas.

```
()
```

```
("ola")
```

```
("ola" ("bom" . "dia") "adeus")
```

```
("ola" (((("pois")))) "então")
```

**Resposta:**

```
> (nova-lista)
()
```

```
> (insere "ola" (nova-lista))
("ola")
```

```
> (insere "ola" (insere (cons "bom" "dia") (insere "adeus" (nova-lista))))
("ola" ("bom" . "dia") "adeus")
```

```
> (insere "ola"
      (insere (insere (insere (insere "pois" (nova-lista))
                             (nova-lista))
                (nova-lista))
              (insere "entao" (nova-lista))))
("ola" (((("pois")))) "entao")
```

Nota: reparar que existe uma lista em que um dos seus elementos é do tipo *par*.

**Exercício 7.4**

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

Note que neste exercício estamos a usar directamente a representação interna das listas em Scheme, o que deveria ser feito usando as operações definidas para o tipo.

```
(list 1 2 3)

(list (cons (integer? 2.0) (rational? 2.0))
      (cons (real? 2.0) (integer? 2/3)))

(cdr (list (string? "ola") (boolean? "ola") 4))

(car (cdr (list 2 3)))

(cdr (cdr (list 2 3)))

()

(list ())

(list)

(pair? (cons 2 3))

(list? (cons 2 3))

(list? (cons 2 ()))

(list? (list 2 3))

(pair? (list 2 3 4))

(null? (list 1 2))

(null? ())

(list? ())

(pair? ())

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

**Resposta:**

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

> (list (cons (integer? 2.0) (rational? 2.0))
        (cons (real? 2.0) (integer? 2/3)))
```

```
(cons (real? 2.0) (integer? 2/3))
((#t . #t) (#t . #f))

> (cdr (list (string? "ola") (boolean? "ola") 4))
(#f 4)

> (car (cdr (list 2 3)))
3

> (cdr (cdr (list 2 3)))
()

> ()
()

> (list ())
(())

> (list)
()

> (pair? (cons 2 3))
#t

> (list? (cons 2 3))
#f

> (list? (cons 2 ()))
#t

> (list? (list 2 3))
#t

> (pair? (list 2 3 4))
#t

> (null? (list 1 2))
#f

> (null? ())
#t

> (list? ())
#t

> (pair? ())
#f

> (null? (list ()))
#f
```

Nota: Em Scheme, todas as listas, à exceção da lista vazia, são pares. Um par é uma lista se o seu segundo elemento for uma lista (logo, uma lista é uma lista se o seu segundo elemento for uma lista). A lista vazia é uma lista.

**Exercício 7.5**

(Adaptado de Sussman — 2.24) Suponha que avaliamos as expressões:

```
(lista 1 (lista 2 (lista 3 4)))
```

```
(insere 1 (insere 2 (lista 3 4)))
```

Para cada uma delas, mostre o resultado impresso pelo interpretador, e a estrutura de caixas e ponteiros correspondente.

**Resposta:**

```
> (lista 1 (lista 2 (lista 3 4)))
(1 (2 (3 4)))
```

```
> (insere 1 (insere 2 (lista 3 4)))
(1 2 3 4)
```

**Exercício 7.6**

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

```
> (ultimo-par (lista 23 72 149 34))
(34)
```

```
> (ultimo-par (nova-lista))
cdr: expects argument of type <pair>; given ()
```

```
> (ultimo-par (lista (nova-lista)))
(())
```

**Resposta:**

```
(define (ultimo-par lst)
  (if (lista-vazia? (resto lst))
      lst
      (ultimo-par (resto lst))))
```

**Exercício 7.7**

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

```
> (inverte (lista 1 4 9 16 25))
(25 16 9 4 1)
```

**Resposta:**

Primeira tentativa:

```
(define (inverte lst)
  (if (lista-vazia? lst)
      (nova-lista)
      (insere (primeiro lst)
              (inverte (resto lst)))))
```

Exemplos:

```
> (inverte (lista 1 4 9 16 25))
(1 4 9 16 25)
```

Mas este procedimento não inverte a lista! Precisamos de ir tirando os elementos do início da lista e ir colocando numa lista auxiliar. Para isso, definimos um procedimento auxiliar, com um argumento adicional, onde vamos guardando os resultados intermédios. Isto tem a vantagem adicional de o procedimento `inverte` passar a gerar um processo iterativo.

```
(define (inverte lst)
  (define (inv-aux lst res)
    (if (lista-vazia? lst)
        res
        (inv-aux (resto lst) (insere (primeiro lst) res))))
  (r-aux lst (nova-lista)))
```

Exemplos:

```
> (inverte (lista 1 4 9 16 25))
(25 16 9 4 1)
```

### Exercício 7.8

Defina um procedimento `junta2`, que recebe como argumento duas listas e retorna uma lista com os elementos de ambas as listas recebidas como argumento:

```
> (junta2 (lista 1 4 9 16 25) (lista 1 2 3 4))
(1 4 9 16 25 1 2 3 4)
```

**Resposta:**

```
(define (junta2 lst1 lst2)
  (if (lista-vazia? lst1)
      lst2
      (insere (primeiro lst1)
              (junta2 (resto lst1) lst2))))
```

Exemplos:

```
> (junta2 (lista 1 4 9 16 25) (lista 1 2 3 4))
(1 4 9 16 25 1 2 3 4)
```

```
> (junta2 (lista 1 2 (lista 3 4)) (lista 5 6))
(1 2 (3 4) 5 6)
```

**Exercício 7.9**

(Sussman — 2.26) Suponha que definimos  $x$  e  $y$  como sendo duas listas:

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

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

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

```
(junta2 x y)
```

```
(insere x y)
```

```
(lista x y)
```

**Resposta:**

```
> (junta2 x y)
(1 2 3 4 5 6)
```

```
> (insere x y)
((1 2 3) 4 5 6)
```

```
> (lista x y)
((1 2 3) (4 5 6))
```

**Exercício 7.10**

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 de inteiros e retorna a soma de todos os elementos dessa lista.
4. O procedimento `aplica-op-com-passo` que recebe uma lista e dois procedimentos e retorna outra lista, cujos elementos são obtidos aplicando o primeiro procedimento ao primeiro elemento da lista inicial, e das sucessivas listas que são obtidas por aplicação do segundo procedimento (até que a lista fique vazia). Por exemplo,

```
> (aplica-op-com-passo (list 1 2 3 4 5) (lambda (x) (* 2 x)) caddr)
(2 6 10)
```

5. 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 (lista (cons "Luisa" 123456789)
                               (cons "Pedro" 234567891)
                               (cons "Maria" 345678912)
                               (cons "Paulo" 456789123)))
```

deverá imprimir

```
Luisa -> 123456789
Pedro -> 234567891
Maria -> 345678912
Paulo -> 456789123
Fim da lista
```

### Resposta:

1. 

```
(define (primeiro-par lst)
  (if (and (not (lista-vazia? lst)) (not (lista-vazia? (resto lst))))
      (cons (primeiro lst) (primeiro (resto lst)))
      (error "primeiro-par: espera lista de pelo menos 2 elems, recebeu"
             lst)))
```

Exemplos:

```
> (primeiro-par (nova-lista))
primeiro-par: espera lista de pelo menos 2 elems, recebeu ()

> (primeiro-par (lista 1))
primeiro-par: espera lista de pelo menos 2 elems, recebeu (1)

> (primeiro-par (lista 1 2 3 4))
(1 . 2)

> (primeiro-par (lista 1 (cons 2 3) 4))
(1 2 . 3)
```

Neste caso, não vamos criar um procedimento que gere um processo recursivo, pois o procedimento `primeiro-par` nem sequer é recursivo.

2. 

```
(define (maior-elemento lst)
  (define (m-e-aux lst)
    (if (lista-vazia? (resto lst))
        (primeiro lst)
        (max (primeiro lst)
              (m-e-aux (resto lst)))))
  (if (or (lista-vazia? lst) (not (lista? lst)))
      (error "maior-elemento: espera lista de pelo menos 1 elem, recebeu"
             lst)
      (m-e-aux lst)))
```

Para o procedimento `maior-elemento` gerar um processo iterativo, usamos o seguinte:

```
(define (maior-elemento lst)
  (define (m-e-aux lst maximo)
    (if (lista-vazia? lst)
        maximo
        (m-e-aux (resto lst) (max (primeiro lst) maximo))))
  (if (or (lista-vazia? lst) (not (lista? lst)))
      (error "maior-elemento: espera lista de pelo menos 1 elem, recebeu"
            lst)
      (m-e-aux (resto lst) (primeiro lst))))
```

Exemplos:

```
> (maior-elemento (nova-lista))
maior-elemento: espera lista de pelo menos 1 elem, recebeu ()
```

```
> (maior-elemento (lista 1 5 2 10 4))
10
```

```
3. (define (soma-elementos lst)
    (if (lista-vazia? lst)
        0
        (+ (primeiro lst)
           (soma-elementos (resto lst)))))
```

Para gerar um processo iterativo, precisamos de um argumento adicional para guardar os resultados intermédios e por causa disso de um procedimento auxiliar

```
(define (soma-elementos lst)
  (define (s-e-aux lst acc)
    (if (lista-vazia? lst)
        acc
        (s-e-aux (resto lst) (+ (primeiro lst) acc))))
  (s-e-aux lst 0))
```

Exemplos:

```
> (soma-elementos (lista 1 5 2 10 4))
22
```

```
4. (define (aplica-op-com-passo l p1 p2)
    (if (null? l)
        ()
        (cons (p1 (car l))
              (aplica-op-com-passo (p2 l) p1 p2))))
```

Exemplos:

```
(aplica-op-com-passo (list 1 2 3 4 5) (lambda (x) (* 2 x)) caddr)
caddr: expects argument of type <caddrable value>; given (5)
```

O problema foi que no último passo da recursão chamámos p2 com (5), o que originou este erro. No entanto, no procedimento aplica-op-com-passo não podemos resolver este problema, pois não sabemos qual vai ser o procedimento passado no terceiro argumento. Assim, quando chamarmos aplica-op-com-passo é que vamos ter o cuidado de passar uma lista que não vá originar um erro. Neste exemplo, a lista deveria ter mais um elemento, mas que não vai ser usado para nada. Por exemplo:

```
(aplica-op-com-passo (list 1 2 3 4 5 ()) (lambda (x) (* 2 x)) caddr)
(2 6 10)
```

Outra alternativa seria que o segundo procedimento testasse se pode ser aplicado, de modo a não dar erro. Por exemplo:

```
(aplica-op-com-passo (list 1 2 3 4 5)
  (lambda (x) (* 2 x))
  (lambda (x) (if (>= (length x) 2)
    (caddr x)
    ())))
```

Para gerar um processo iterativo, precisamos de um argumento auxiliar e por causa disso de um procedimento auxiliar. Uma possibilidade é:

```
(define (aplica-op-com-passo l p1 p2)
  (define (a-o-c-p-aux l res)
    (if (null? l)
      res
      (a-o-c-p-aux (p2 l) (cons (p1 (car l)) res))))
  (a-o-c-p-aux l ()))
```

Exemplos:

```
(aplica-op-com-passo (list 1 2 3 4 5 ()) (lambda (x) (* 2 x)) caddr)
(10 6 2)
```

O problema é que assim ficamos com a lista de resultados invertida! Não é trivial criar um processo iterativo neste caso, pois não sabemos acrescentar elementos no fim de listas. Para isso deveríamos definir um outro procedimento, mas ficaria muito pouco eficiente porque tinha que percorrer sempre toda a lista, e assim iríamos perder as vantagens dos processos iterativos.

```
5. (define (imprime-lista-de-pares lst)
  (cond ((lista-vazia? lst) (display "Fim da lista"))
        ((not (pair? (primeiro lst)))
         (error "imprime-lista-de-pares: espera lista de pares, recebeu"
                lst))
        (else (display (car (primeiro lst)))
              (display " -> ")
              (display (cdr (primeiro lst)))
              (newline)
              (imprime-lista-de-pares (resto lst)))))
```

Exemplos:

```
> (imprime-lista-de-pares (lista (cons "Luisa" 123456789)
  (cons "Pedro" 234567891)
  (cons "Maria" 345678912)
  (cons "Paulo" 456789123)))
```

```
Luisa -> 123456789
Pedro -> 234567891
Maria -> 345678912
Paulo -> 456789123
Fim da lista
```

```
> (imprime-lista-de-pares (lista (cons "Luisa" 123456789)
  (cons "Pedro" 234567891)
  (cons "Maria" 345678912)
  "Paulo"))
```

```
Luisa -> 123456789
Pedro -> 234567891
Maria -> 345678912
imprime-lista-de-pares: espera lista de pares, recebeu ("Paulo")
```

Este exemplo gera um processo iterativo e neste caso não faz sentido criar um procedimento que gere um processo recursivo.

## 8 Funcionais Sobre Listas

### Sumário:

- Funcionais sobre listas

### Resumo:

- Funcionais sobre listas são procedimentos de ordem superior que efectuam operações sobre listas.
  - Um *transformador* é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e devolve uma lista em que cada elemento resulta da aplicação da operação ao elemento correspondente da lista original.
  - Um *filtro* é um funcional que recebe como argumentos uma lista e um predicado aplicável aos elementos da lista, e devolve a lista constituída apenas pelos elementos da lista original que satisfazem o predicado.
  - Um *acumulador* é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e aplica sucessivamente essa operação aos elementos da lista original, devolvendo o resultado da aplicação a todos os elementos da lista.
- `map`, `for-each`, `length`
- Definição alternativa para o tipo Lista  
Construtores

```
(define (nova-lista) ())
(define insere cons)
```

### Selectores

```
(define primeiro car)
(define resto cdr)
```

### Reconhecedores

```
(define lista? list?)
(define lista-vazia? null?)
```

### Testes

```
(define listas=? equal?)
```

### Alto nível

```
(define lista list)
```

**Exercício 8.1**

Defina um procedimento `comprimento`, que recebe como argumento uma lista, e retorna um inteiro correspondente ao número de elementos dessa lista.

```
> (comprimento (lista -10 2.5 -11.6 17))
4
```

```
> (comprimento (lista (nova-lista)))
1
```

```
> (comprimento (nova-lista))
0
```

**Resposta:**

```
(define (comprimento lst)
  (if (lista-vazia? lst)
      0
      (+ 1 (comprimento (resto lst)))))
```

**Exercício 8.2**

Defina um procedimento `mapeia`, 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.

```
> (mapeia abs (lista -10 2.5 -11.6 17))
(10 2.5 11.6 17)
```

```
> (mapeia (lambda (x) (* x 5)) (lista 1 2 3 4 5))
(5 10 15 20 25)
```

**Resposta:**

```
(define (mapeia proc lst)
  (if (lista-vazia? lst)
      (nova-lista)
      (insere (proc (primeiro lst))
              (mapeia proc (resto lst)))))
```

**Exercício 8.3**

(Sussman — 2.23) O procedimento `para-cada` é semelhante ao `mapeia`. Recebe como argumentos um procedimento e uma lista de elementos. No entanto, em vez de formar uma lista com os resultados, `para-cada` 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 — `para-cada` é usado com procedimentos que executam uma ação, tal como imprimir. Por exemplo:

```
> (para-cada (lambda (x) (newline) (display x))
      (lista 57 321 28))
57
321
28
```

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

**Resposta:**

```
(define (para-cada proc lst)
  (cond ((lista-vazia? lst) #t)
        (else (proc (primeiro lst))
                (para-cada proc (resto lst)))))
```

**Exemplos:**

```
> (para-cada (lambda (x) (newline) (display x))
      (lista 57 321 28))
57
321
28#t
```

#### Exercício 8.4

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 (lista 1 2 3 2 5))
(1 3 3 3 5)

> (substitui 2 3 (lista 1 3 5 7))
(1 3 5 7)
```

**Resposta:**

```
(define (substitui velho novo lst)
  (cond ((lista-vazia? lst) (nova-lista))
        ((= (primeiro lst) velho)
         (insere novo
                  (substitui velho novo (resto lst))))
        (else (insere (primeiro lst)
                       (substitui velho novo (resto lst)))))
```

#### Exercício 8.5

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

**Resposta:**

```
(define (substitui velho novo lst)
  (mapeia (lambda (elem)
           (if (eq? elem velho)
               novo
               elem))
          lst))
```

**Exercício 8.6**

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

```
> (quadrado-lista (lista 1 2 3 4))
(1 4 9 16)
```

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

```
(define (quadrado-lista lst)
  (if (lista-vazia? lst)
      (nova-lista)
      (insere <??> <??>)))

(define (quadrado-lista lst)
  (mapeia <??> <??>))
```

**Resposta:**

```
(define (quadrado-lista lst)
  (if (lista-vazia? lst)
      (nova-lista)
      (insere (* (primeiro lst) (primeiro lst))
              (quadrado-lista (resto lst)))))
```

Uma alternativa, usando uma `lambda`, é:

```
(define (quadrado-lista lst)
  (if (lista-vazia? lst)
      (nova-lista)
      (insere (* (primeiro lst) (primeiro lst))
              (quadrado-lista (resto lst)))))
```

Usando o procedimento `mapeia`, fica:

```
(define (quadrado-lista lst)
  (mapeia (lambda (x) (* x x)) lst))
```

**Exercício 8.7**

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? (lista 1 2 3 4 5))
(2 4)

> (filtra even? (lista 1 3 5 7))
()
```

**Resposta:**

```
(define (filtra pred lst)
  (cond ((lista-vazia? lst) (nova-lista))
        ((pred (primeiro lst))
         (insere (primeiro lst)
                 (filtra pred (resto lst))))
        (else (filtra pred (resto lst)))))
```

**Exercício 8.8**

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? (lista 1 2 3 4 5))
#f

> (todos? even? (lista 2 4 6))
#t
```

**Resposta:**

```
(define (todos? pred lst)
  (cond ((lista-vazia? lst) #t)
        ((pred (primeiro lst))
         (todos? pred (resto lst)))
        (else #f)))
```

**Exercício 8.9**

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? (lista 1 2 3 4 5))
#t

> (algum? odd? (lista 2 4 6))
#f
```

**Resposta:**

```
(define (algun? pred lst)
  (cond ((lista-vazia? lst) #f)
        ((pred (primeiro lst)) #t)
        (else (algun? pred (resto lst)))))
```

**Exercício 8.10**

Escreva um procedimento `prega-direita` 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:

```
> (prega-direita + 0 (lista 1 2 3 4))
10

> (prega-direita + 0 (nova-lista))
0

> (prega-direita string-append "" (lista "ola" "bom" "dia"))
"olabomdia"
```

**Resposta:**

```
(define (prega-direita proc inicial lst)
  (if (lista-vazia? lst)
      inicial
      (proc (primeiro lst)
            (prega-direita proc inicial (resto lst)))))
```

**Exercício 8.11**

Com base no procedimento `prega-direita` 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. `junta2` que recebe duas listas e retorna outra lista que resulta de juntar as duas listas iniciais.

**Resposta:**

1. 

```
(define (multiplica-lista lst)
  (if (lista-vazia? lst)
      (error "multiplica-lista: espera lista com pelo menos 1 elem, recebeu"
            lst)
```

```

      (prega-direita * 1 lst)))

> (multiplica-lista (lista 1 2 3 4))
24

> (multiplica-lista (nova-lista))
multiplica-lista: espera lista com pelo menos 1 elem, recebeu ()

2. (define (maximo-lista lst)
    (if (lista-vazia? lst)
        (error "maximo-lista: espera lista com pelo menos 1 elem, recebeu"
              lst)
        (prega-direita max (primeiro lst) (resto lst))))

> (maximo-lista (lista 1 2 3 4))
4

> (maximo-lista (nova-lista))
maximo-lista: espera lista com pelo menos 1 elem, recebeu ()

3. (define (inverte-lista lst)
    (prega-direita (lambda (x y) (junta2 y (lista x))) (nova-lista) lst))

> (inverte-lista (lista 1 2 3 4))
(4 3 2 1)

> (inverte-lista (nova-lista))
()

4. (define (junta2 lst1 lst2)
    (prega-direita insere lst2 lst1))

> (junta2 (lista 1 2 3 4) (lista 5 6 7))
(1 2 3 4 5 6 7)

> (junta2 (nova-lista) (lista 5 6 7))
(5 6 7)

```

### Exercício 8.12

(Sussman — 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 (subconjuntos c)
  (if (lista-vazia? c)
      (lista (nova-lista))
      (let ((resto-c (subconjuntos (resto c))))
        (junta resto-c (mapeia <??> resto-c)))))

```

### Resposta:

```

(define (subconjuntos c)

```

```
(if (lista-vazia? c)
    (lista (nova-lista))
    (let ((resto-c (subconjuntos (resto c))))
        (junta resto-c (mapeia (lambda (x) (insere (primeiro c) x)) resto-c))))))
```

**Exemplos:**

```
> (subconjuntos (lista))
(())

> (subconjuntos (lista 1 2 3))
(()) (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

### Exercício 8.13

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

```
> (conta-se even? '(1 2 3 2 5))
2
```

**Resposta:**

```
(define (conta-se pred lst)
  (if (lista-vazia? lst)
      0
      (+ (if (pred (primeiro lst)) 1 0)
         (conta-se pred (resto lst)))))
```

### Exercício 8.14

Escreva um procedimento `posicao` 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 0 (zero). Por exemplo:

```
> (posicao 'a '(a b c a b))
0

> (posicao 'a '(b c a b))
2

> (posicao 'd '(b c a b))
#f
```

**Resposta:**

```
(define (posicao elem lst)
  (if (lista-vazia? lst)
```

```
#f
(let ((pos (posicao elem (resto lst))))
  (if (number? pos)
      (+ 1 pos)
      pos))))
```

**Exercício 8.15**

Usando os procedimentos `filtra`, `curry` e `complement` escreva um procedimento `remove` que recebe um 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)
```

**Resposta:**

```
(define (remove elem lst)
  (filtra (complement (curry eq? elem)) lst))
```

**Exercício 8.16**

Escreva um procedimento `junta-unico` 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:

```
> (junta-unico 'a '(a b c))
(a b c)

> (junta-unico 'd '(a b c))
(d a b c)
```

**Resposta:**

```
(define (junta-unico elem lst)
  (if (membro-eq elem lst)
      lst
      (insere elem lst)))
```

**Exercício 8.17**

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

```
> (remove-duplicados '(a b c))
(a b c)

> (remove-duplicados '(a b a a c a))
(b c a)
```

**Resposta:**

```
(define (remove-duplicados lst)
  (prega-direita junta-unico (nova-lista) lst))
```

### Exercício 8.18

Escreva um procedimento `junta-ordenado` que recebe duas listas de números por ordem crescente e devolve uma lista com todos os números das duas listas ordenados por ordem crescente. Por exemplo:

```
(junta-ordenado '(1 4 7 10) '(2 4 5 12))
(1 2 4 4 5 7 10 12)
```

**Resposta:**

```
(define (junta-ordenado lst1 lst2)
  (cond ((lista-vazia? lst1) lst2)
        ((lista-vazia? lst2) lst1)
        (< (primeiro lst1) (primeiro lst2))
        (insere (primeiro lst1) (junta-ordenado (resto lst1) lst2)))
  (else (insere (primeiro lst2) (junta-ordenado lst1 (resto lst2))))))
```

### Exercício 8.19

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 `codifica-lista` que compacta listas de inteiros. Por exemplo,

```
> (codifica-lista '(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))

> (codifica-lista '(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 `descodifica-lista` que, dada uma lista de inteiros compactada, retorna a lista original. Por exemplo,

```
> (descodifica-lista '((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)

> (descodifica-lista '(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)
```

**Resposta:**

```

(define (codifica-lista lst)
  (define (codifica-seq lst ultimo contador)
    (define (faz-codigo ultimo contador)
      (if (> contador 1)
          (insere contador ultimo)
          ultimo))
    (cond ((lista-vazia? lst) (lista (faz-codigo ultimo contador)))
          ((= ultimo (primeiro lst))
           (codifica-seq (resto lst) ultimo (add1 contador)))
          (else (insere (faz-codigo ultimo contador)
                        (codifica-seq (resto lst)
                                      (primeiro lst)
                                      1))))))
  (if (lista-vazia? lst)
      (nova-lista)
      (codifica-seq (resto lst) (primeiro lst) 1)))

(define (descodifica-lista lst)
  (define (descodifica codigo)
    (define (make-lista contador elem)
      (if (zero? contador)
          (nova-lista)
          (insere elem (make-lista (sub1 contador) elem))))
    (if (pair? codigo)
        (make-lista (primeiro codigo) (resto codigo))
        (lista codigo)))
  (if (lista-vazia? lst)
      (nova-lista)
      (junta (descodifica (primeiro lst))
             (descodifica-lista (resto lst)))))

```





### O tipo *símbolo*

- A criação de símbolos é feita automaticamente pelo avaliador do Scheme ao ler uma expressão.
  - *Construtores*: `string->symbol`
  - *Selectores*: não tem porque é um tipo elementar.
  - *Reconhecedores*: `symbol?`
  - *Testes*: `eq?`
- A forma especial `quote` não avalia os argumentos, tendo como valor o próprio argumento.
- Abreviatura para `(quote XXX)` é `'XXX`.
- `quote`, `'`

### O tipo *árvore*

- Uma *árvore* é uma estrutura hierárquica composta por uma raiz que domina outras árvores.
- Uma *árvore binária* ou é vazia ou é constituída por uma raiz que domina duas árvores binárias, a árvore esquerda e a árvore direita. No livro, uma árvore binária é representada por um par cujo primeiro elemento contém a raiz da árvore e cujo segundo elemento contém um par com a árvore esquerda e com a árvore direita da árvore inicial; uma árvore vazia é representada por uma lista vazia.
  - *Construtores*: `nova-arv`, `cria-arv`
  - *Selectores*: `raiz`, `arv-esq`, `arv-dir`
  - *Reconhecedores*: `arvore?`, `arvore-vazia?`
  - *Testes*: `arvores=?`

**Exercício 9.1**

(Sussman — 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:

```
(lista a b)
```

```
(lista 'a 'b)
```

```
(lista 'a b)
```

```
(primeiro '(a b c))
```

```
(resto '(a b c))
```

**Resposta:**

```
> (lista a b)
(1 2)
```

```
> (lista 'a 'b)
(a b)
```

```
> (lista 'a b)
(a 2)
```

```
> (primeiro '(a b c))
a
```

```
> (resto '(a b c))
(b c)
```

**Exercício 9.2**

(Sussman — exemplo da página 144) Defina o procedimento `membro-eq`, 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,

```
> (membro-eq 'manga '(pera banana uva))
#f
```

```
> (membro-eq 'manga '(x (manga lima) y manga pera))
(manga pera)
```

**Resposta:**

```
(define (membro-eq elem lst)
  (cond ((lista-vazia? lst) #f)
        ((eq? elem (primeiro lst)) lst)
        (else (membro-eq elem (resto lst)))))
```

**Exercício 9.3**

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

```
(lista 'a 'b 'c)

(lista (lista 'paulo))

(resto '((x1 x2) (y1 y2)))

(primeiro (resto '((x1 x2) (y1 y2))))

(lista? (primeiro '(uma lista curta)))

(lista? (resto '(uma lista curta)))

(membro-eq 'vermelho '((vermelho amarelo) (azul verde)))

(membro-eq 'vermelho '(vermelho amarelo azul verde))
```

**Resposta:**

```
> (lista 'a 'b 'c)
(a b c)

> (lista (lista 'paulo))
((paulo))

> (resto '((x1 x2) (y1 y2)))
((y1 y2))

> (primeiro (resto '((x1 x2) (y1 y2))))
(y1 y2)

> (lista? (primeiro '(uma lista curta)))
#f

> (lista? (resto '(uma lista curta)))
#t

> (membro-eq 'vermelho '((vermelho amarelo) (azul verde)))
#f

> (membro-eq 'vermelho '(vermelho amarelo azul verde))
(vermelho amarelo azul verde)
```

**Exercício 9.4**

(Sussman — 2.55) O resultado de avaliar a expressão

```
(primeiro ''abracadabra)
```

é quote. Explique porquê.

**Resposta:**

Como 'x é uma abreviatura sintáctica para (quote x), escrever (primeiro "abracadabra) é uma abreviatura para (primeiro (quote (quote abracadabra))).

Para avaliar esta expressão:

1. Avaliamos o operador. Neste caso, primeiro, que tem como resultado o procedimento que dado uma lista retorna o seu primeiro elemento.
2. Avaliamos os seus argumentos, por qualquer ordem. Neste caso, avaliar a expressão (quote (quote abracadabra)) tem como resultado a lista (quote abracadabra).
3. Aplicamos o operador aos valores dos argumentos e retornamos o resultado. Neste caso, aplicar o procedimento primeiro à lista (quote abracadabra) tem como resultado o primeiro elemento da lista, que é o símbolo quote.

**Exercício 9.5**

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

```
(listas=? '(isto e uma lista) '(isto e uma lista))
```

é verdade, mas

```
(listas=? '(isto e uma lista) '(isto (e uma) lista))
```

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

**Resposta:**

```
(define (listas=? lst1 lst2)
  (cond ((or (symbol? lst1) (symbol? lst2)) (eq? lst1 lst2))
        ((lista-vazia? lst1) (lista-vazia? lst2))
        ((lista-vazia? lst2) #f)
        (else (and (listas=? (primeiro lst1) (primeiro lst2))
                    (listas=? (resto lst1) (resto lst2))))))
```

```
> (listas=? '(isto e uma lista) '(isto e uma lista))
#t
```

```
> (listas=? '(isto e uma lista) '(isto (e uma) lista))
#f
```

```
> (listas=? '(2 3 a 4) '(2 3 a 4))
car: expects argument of type <pair>; given 2
```

Nota: uma vez que nem os números nem as strings são símbolos, esta definição não funciona para listas com elementos destes tipos. Na realidade, esta definição só funciona para listas de símbolos.

### Exercício 9.6

Considere a definição tipo *árvore binária* à custa das seguintes operações:

- `nova-arv` que constrói uma árvore binária vazia.
- `cria-arv` que recebe a raiz, a árvore binária esquerda e a árvore binária direita e constrói a árvore binária correspondente.
- `raiz` que recebe uma árvore binária e retorna a sua raiz.
- `arv-esq` que recebe uma árvore binária e retorna a sua árvore binária esquerda.
- `arv-dir` que recebe uma árvore binária e retorna a sua árvore binária direita.
- `arvore?` que recebe um objecto e retorna verdadeiro se ele corresponder a uma árvore binária e falso caso contrário.
- `arvore-vazia?` que recebe um objecto e retorna verdadeiro se ele corresponder a uma árvore binária vazia e falso caso contrário.
- `arvores=?` que recebe duas árvores binárias e retorna verdadeiro se elas forem iguais e falso caso contrário.

1. Especifique formalmente estas operações, e classifique-as em construtores, selectores, reconhecedores, e testes.
2. Implemente estas operações em Scheme, usando como representação para as árvores binárias um par em que o primeiro elemento aponta para a raiz e o segundo elemento aponta para outro par, em que o primeiro elemento aponta para a árvore esquerda da árvore original e o segundo elemento aponta para a árvore direita da árvore original.
3. Com base nas operações descritas, escreva os seguintes procedimentos para percorrer árvores binárias:
  - (a) `percorre-inorder` recebe uma árvore binária e retorna uma lista com todos os seus elementos, percorrendo primeiro a árvore esquerda, depois a raiz e depois a árvore direita da árvore inicial.
  - (b) `percorre-preorder` recebe uma árvore binária e retorna uma lista com todos os seus elementos, percorrendo primeiro a raiz, depois a árvore esquerda e depois a árvore direita da árvore inicial.
  - (c) `percorre-posorder` recebe uma árvore binária e retorna uma lista com todos os seus elementos, percorrendo primeiro a árvore esquerda, depois a árvore direita e depois a raiz da árvore inicial.
4. 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, escreva os seguintes procedimentos:

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

**Resposta:**

1. Especificação das operações do tipo

- Construtores:

```
nova-arv: -> Arvore
cria-arv: Elemento x Arvore x Arvore -> Arvore
```

- Selectores:

```
raiz: Arvore -> Elemento
arv-esq: Arvore -> Arvore
arv-dir: Arvore -> Arvore
```

- Reconhecedores:

```
arvore?: Universal -> Lógico
arvore-vazia?: Arvore -> Lógico
```

- Testes

```
arvores=?: Arvore x Arvore -> Lógico
```

2. Definição das operações do tipo. Nesta definição apresentam-se algumas operações que não foram pedidas no enunciado mas que aparecem no livro da cadeira.

```
(define (nova-arv)
  ())

(define (cria-arv raiz arv-esq arv-dir)
  (cons raiz (cons arv-esq arv-dir)))

(define (raiz arv)
  (if (null? arv)
      (error "raiz: a arvore e vazia")
      (car arv)))

(define (arv-esq arv)
  (if (null? arv)
      (error "arv-esq: a arvore e vazia")
      (car (cdr arv))))

(define (arv-dir arv)
  (if (null? arv)
      (error "arv-dir: a arvore e vazia")
      (cdr (cdr arv))))

(define (arvore? x)
  (cond ((null? x) #t)
```

```

      ((and (pair? x) (pair? (cdr x)))
       (and (arvore? (car (cdr x)))
            (arvore? (cdr (cdr x)))))
      (else #f)))

(define (arvore-vazia? arv)
  (null? arv))

(define (arvores=? arv1 arv2 elem=?)
  (cond ((arvore-vazia? arv1) (arvore-vazia? arv2))
        ((arvore-vazia? arv2) #f)
        ((elem=? (raiz arv1) (raiz arv2))
         (and (arvores=? (arv-esq arv1)
                          (arv-esq arv2)
                          elem=?)
              (arvores=? (arv-dir arv1)
                          (arv-dir arv2)
                          elem=?)))
        (else #f)))

(define (escreve-arv arv)
  (define (esc-arv-aux arv ident)
    (cond ((arvore-vazia? arv) (escreve "()" ident))
          (else (escreve (raiz arv) ident)
                (esc-arv-aux (arv-esq arv) (+ ident 2))
                (esc-arv-aux (arv-dir arv) (+ ident 2)))))
  (esc-arv-aux arv 0))

(define (escreve texto ident)
  (define (esc-brancos n)
    (cond ((= n 0))
          (else (display " ")
                (esc-brancos (- n 1)))))
  (newline)
  (esc-brancos ident)
  (display texto))

(define (ordena-lista lst)
  (percorre (lista->arvore lst)))

(define (lista->arv lst)
  (insere-arv lst (nova-arv)))

(define (insere-arv lst arv)
  (if (lista-vazia? lst)
      arv
      (insere-arv (resto lst)
                  (insere-elemento (primeiro lst)
                                    arv))))

(define (insere-elemento elem arv)
  (cond ((arvore-vazia? arv)
        (cria-arv elem (nova-arv) (nova-arv)))
        (> elem (raiz arv))
        (cria-arv (raiz arv)
                  (insere-elemento elem (arv-dir arv)
                                    arv))))

```



```

> (escreve-arv
  (substitui-arv-bin
    2
    8
    (cria-arv 4
      (cria-arv 5
        (nova-arv)
        (cria-arv 2 (nova-arv) (nova-arv)))
      (cria-arv 2
        (cria-arv 3 (nova-arv) (nova-arv))
        (nova-arv))))))
4
5
  ()
  8
    ()
    ()
  8
    3
    ()
    ()
  ()

```

**Resposta:**

```

(define (substitui-arv-bin velho novo arv)
  (cond ((arvore-vazia? arv) arv)
        (else (cria-arv (if (= (raiz arv) velho) novo (raiz arv))
                        (substitui-arv-bin velho novo (arv-esq arv))
                        (substitui-arv-bin velho novo (arv-dir arv))))))

```

**Exercício 9.8**

Escreva um procedimento `profundidade-arv-bin` que recebe uma árvore binária e devolve um número que indica qual é o nível mais profundo de folhas dentro dessa árvore, aumentando de um sempre que se entra para dentro de uma árvore. Se a árvore é vazia, a sua profundidade é 0. Por exemplo:

```

> (profundidade-arv-bin (nova-arv))
0

> (profundidade-arv-bin
  (cria-arv 4
    (cria-arv 5
      (nova-arv)
      (nova-arv))
    (nova-arv)))
2

```

```
> (profundidade-arv-bin
   (cria-arv 4
     (cria-arv 5
       (nova-arv)
       (cria-arv 2
         (nova-arv)
         (nova-arv)))
     (cria-arv 2
       (cria-arv 3
         (nova-arv)
         (nova-arv))
       (nova-arv))))
```

3

**Resposta:**

```
(define (profundidade-arv-bin arv)
  (cond ((arvore-vazia? arv) 0)
        (else (+ 1 (max (profundidade-arv-bin (arv-esq arv))
                        (profundidade-arv-bin (arv-dir arv)))))))
```

**Exercício 9.9**

Escreva um procedimento `conta-elems-arv-bin` que recebe uma árvore binária e devolve um número que indica qual é o número de elementos que existem nessa árvore. Se a árvore é vazia, tem 0 elementos. Por exemplo:

```
> (conta-elems-arv-bin (nova-arv))
0
```

```
> (conta-elems-arv-bin
   (cria-arv 4
     (cria-arv 5
       (nova-arv)
       (nova-arv))
     (nova-arv)))
```

2

```
> (conta-elems-arv-bin
   (cria-arv 4
     (cria-arv 5
       (nova-arv)
       (cria-arv 2
         (nova-arv)
         (nova-arv)))
     (cria-arv 2
       (cria-arv 3
         (nova-arv)
         (nova-arv))
       (nova-arv))))
```

```
(nova-arv)))
```

5

**Resposta:**

```
(define (conta-elems-arv-bin arv)
  (cond ((arvore-vazia? arv) 0)
        (else (+ 1
                  (conta-elems-arv-bin (arv-esq arv))
                  (conta-elems-arv-bin (arv-dir arv))))))
```

**Exercício 9.10**

Escreva um procedimento `inverte-arv-bin` que recebe uma árvore binária como argumento e retorna uma outra árvore binária em que as suas árvore esquerda e árvore direita estão trocadas, bem como as de todas as suas sub-árvores. Por exemplo,

```
> (escreve-arv
   (inverte-arv-bin (nova-arv)))
()
```

```
> (escreve-arv
   (inverte-arv-bin (cria-arv 4
                          (cria-arv 5
                                      (nova-arv)
                                      (nova-arv))
                          (nova-arv))))
```

4

()

5

()

()

```
> (escreve-arv
   (inverte-arv-bin (cria-arv 4
                          (cria-arv 5
                                      (nova-arv)
                                      (cria-arv 2 (nova-arv) (nova-arv)))
                          (cria-arv 2
                                      (cria-arv 3 (nova-arv) (nova-arv))
                                      (nova-arv))))))
```

4

2

()

3

()

()

5

2

```

    ()
  ()
()

```

**Resposta:**

```

(define (inverte-arv-bin arv)
  (cond ((arvore-vazia? arv) arv)
        (else (cria-arv (raiz arv)
                        (inverte-arv-bin (arv-dir arv))
                        (inverte-arv-bin (arv-esq arv))))))

```

**Exercício 9.11**

Escreva um procedimento `alisa-arv-bin` que recebe como argumento uma árvore binária e retorna uma lista cujos elementos são todos os elementos da árvore, da esquerda para a direita. Por exemplo,

```

> (alisa-arv-bin (nova-arv))
()

> (alisa-arv-bin (cria-arv 4
                        (cria-arv 5
                                (nova-arv)
                                (nova-arv))
                        (nova-arv)))
(5 4)

> (alisa-arv-bin (cria-arv 4
                        (cria-arv 5
                                (nova-arv)
                                (cria-arv 2 (nova-arv) (nova-arv)))
                        (cria-arv 2
                                (cria-arv 3 (nova-arv) (nova-arv))
                                (nova-arv))))
(5 2 4 3 2)

```

**Resposta:**

```

(define (alisa-arv-bin arv)
  (cond ((arvore-vazia? arv) arv)
        (else (junta (alisa-arv-bin (arv-esq arv))
                    (lista (raiz arv))
                    (alisa-arv-bin (arv-dir arv))))))

```

Nota: este exercício é semelhante ao `percorre-inorder`.

**Exercício 9.12**

Escreva um procedimento `mapeia-arv`, semelhante ao `mapeia`, que recebe um procedimento de um argumento e uma árvore e retorna outra árvore em que cada elemento

corresponde à aplicação do procedimento ao elemento correspondente da árvore original. Com base nesse procedimento escreva o procedimento `mult-3-arv` que recebe uma árvore como argumento e tem como resultado outra árvore em que cada elemento corresponde ao elemento da árvore original multiplicado por 3. Por exemplo:

```
> (escreve-arv
   (mult-3-arv-bin (nova-arv)))
()

> (escreve-arv
   (mult-3-arv-bin (cria-arv 3
                    (nova-arv)
                    (nova-arv))))
9
()
()

> (escreve-arv
   (mult-3-arv-bin (cria-arv 4
                           (cria-arv 5
                                       (nova-arv)
                                       (cria-arv 2 (nova-arv) (nova-arv)))
                           (cria-arv 2
                                       (cria-arv 3 (nova-arv) (nova-arv))
                                       (nova-arv))))))
12
15
()
6
()
()
6
9
()
()
()
```

**Resposta:**

```
(define (mapeia-arv-bin proc arv)
  (cond ((arvore-vazia? arv) arv)
        (else (cria-arv (proc (raiz arv))
                        (mapeia-arv-bin proc (arv-esq arv))
                        (mapeia-arv-bin proc (arv-dir arv))))))

(define (mult-3-arv-bin arv)
  (mapeia-arv-bin (lambda (x) (* x 3)) arv))
```

## 10 Programação Imperativa

### Sumário:

- Programação Imperativa
- Procedimentos com Estado Interno

### Resumo:

- Um objecto tem *estado* se o seu comportamento é influenciado pela sua história.
- O estado de uma entidade é caracterizado por uma ou mais *variáveis de estado*, as quais mantêm informação sobre a sua história, de modo a poder determinar o comportamento da entidade.
- <operação de atribuição> ::= (set! <nome> <expressão>)
- Ao introduzir a atribuição, duas chamadas ao mesmo procedimento com os mesmos argumentos podem produzir resultados diferentes.
- Por convenção, os nomes dos procedimentos que usam o *set* vão terminar em *!*.
- Um *efeito* corresponde à alteração de qualquer entidade associada a um processo computacional. Os procedimentos baseados em efeitos não são avaliados pelo valor que produzem, mas sim pelo efeito que originam.
- A programação *imperativa* baseia-se no conceito de efeito. Um programa é considerado como uma sequência de instruções, cada uma das quais produz um efeito.
- Na programação imperativa, a *ordem* de execução das instruções é crucial para o bom funcionamento do programa.
- *set!*

### O tipo *Caixa*

- Uma caixa tem um e um só valor. Por esta razão, não podem existir caixas sem valor, e quando é colocado um novo valor numa caixa o anterior perde-se.
- Uma caixa é um tipo de informação mutável. Para alterar os valores dos elementos destes tipos existem os modificadores.
  - *Construtores*: *box*
  - *Selectores*: *unbox*
  - *Modificadores*: *set-box!*
  - *Reconhecedores*: *box?*
  - *Testes*: *equal?*

### Métodos de passagem de parâmetros

- **Passagem por valor** — Quando um parâmetro é passado por valor, o valor do parâmetro concreto é avaliado e esse valor é associado com o parâmetro formal correspondente.
- **Passagem por referência** — Quando um parâmetro é passado por referência, o que é associado ao parâmetro formal correspondente é o “contentor” que contém o valor do parâmetro concreto. A maneira de simular este método em Scheme é através da utilização de caixas.

### **Procedimentos com Estado Interno**

- Os procedimentos com estado interno mantêm encapsuladas variáveis internas. De cada vez que o procedimento é chamado, é criado um novo ambiente, em que as variáveis internas são criadas com os seus valores iniciais.

**Exercício 10.1**

Diga o que é impresso pelo interpretador de Scheme ao avaliar cada uma das seguintes expressões, supondo que elas são avaliadas pela ordem apresentada. Se alguma delas der origem a um erro, explique porquê.

```
(define a 3)

(set! a "ola")

(+ a 1)

(string-append a " bom dia")

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

(set! c 78)
```

**Resposta:**

```
> (define a 3)

> (set! a "ola")

> (+ a 1)
+!: expects type <number> as 1st argument, given: "ola"; other arguments were: 1

> (string-append a " bom dia")
"ola bom dia"

> (begin
  (let ((a 5))
    (+ a (* 45 327))
    (sqrt (length '(1 a b "bom dia" (2 5) 3))))
  (display "a\nb\n")
  a)
a
b
"ola"

> (set! c 78)
set!: cannot set undefined identifier: c
```

**Exercício 10.2**

Introduzir a forma especial `set!` na nossa linguagem obriga-nos a pensar no significado de igualdade e mudança. Considere as definições dos procedimentos:

```
(define (cria-somaA inicial)
```

```
(lambda (num)
  (+ inicial num)))

(define (cria-somaB! inicial)
  (lambda (num)
    (set! inicial (+ inicial num))
    inicial))
```

No caso do `cria-somaA`, várias chamadas ao procedimento com o mesmo argumento dão o mesmo resultado.

No caso do `cria-somaB!`, várias chamadas ao procedimento com o mesmo argumento podem dar resultados diferentes.

Através de exemplos de chamadas a estes procedimentos, justifique estas duas frases.

**Resposta:**

Exemplos para `cria-somaA`:

```
> (define sA1 (cria-somaA 25))
> (define sA2 (cria-somaA 25))

> (sA1 20)
45

> (sA1 20)
45

> (sA2 20)
45
```

Neste caso, podemos dizer que `sA1` e `sA2` são iguais, pois um pode ser substituído pelo outro em qualquer lugar da computação sem alterar o resultado. Uma linguagem que suporta o conceito de que “iguais podem ser substituídos por iguais” numa expressão sem alterar o valor da expressão diz-se *referencialmente transparente*.

A transparência referencial é violada quando introduzimos o `set!` na linguagem, o que dificulta a verificação de quando é que podemos simplificar expressões substituindo-as por outras equivalentes.

Exemplos para `cria-somaB!`:

```
> (define sB1 (cria-somaB! 25))
> (define sB2 (cria-somaB! 25))

> (sB1 20)
45

> (sB1 20)
65

> (sB2 20)
45
```

**Exercício 10.3**

(Sussman — 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 `faz-acumulador` que gera acumuladores, cada um dos quais mantendo uma soma independente. O valor de entrada para o procedimento `faz-acumulador` deve especificar o valor inicial da soma. Por exemplo,

```
> (define a (faz-acumulador 5))
```

```
> (a 10)
```

```
15
```

```
> (a 10)
```

```
25
```

**Resposta:**

```
(define (faz-acumulador inicial)
  (lambda (num)
    (set! inicial (+ inicial num))
    inicial))
```

**Exercício 10.4**

O Rui e o Nuno têm que completar 100 créditos de disciplinas para terminarem os respectivos cursos. Mostre como é que o procedimento do exercício anterior poderia ser usado para manter o registo de créditos de cada um dos alunos.

**Resposta:**

```
> (define ac-rui (faz-acumulador 0))
```

```
> (define ac-nuno (faz-acumulador 0))
```

```
> (ac-rui 20)
```

```
20
```

```
> (ac-nuno 10)
```

```
10
```

```
> (ac-rui 20)
```

```
40
```

```
> (ac-nuno 20)
```

```
30
```

```
> (ac-rui -5)
```

```
35
```

Nota: cada acumulador que é criado tem o seu próprio contador, e não tem qualquer influência nos contadores dos outros acumuladores.

**Exercício 10.5**

(Sussman — 3.2) Em aplicações para testar software, é útil ser capaz de contar o número de vezes que um procedimento é chamado no decurso de uma computação.

Escreva um procedimento `faz-monitor` que recebe um procedimento `f` como argumento, que por sua vez é um procedimento de um argumento. O resultado retornado pelo procedimento `faz-monitor` é um terceiro procedimento `fm` que mantém um registo do número de vezes que foi chamado através de um contador interno. Se o valor de entrada para `fm` for o símbolo `quantas-chamadas?`, então `fm` deve retornar o valor do contador. Se o valor de entrada for o símbolo `inicializa-contador`, então `fm` deve inicializar o contador a zero. Para qualquer outro valor de entrada, `fm` 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 (faz-monitor sqrt))

> (s 100)
10

> (s 'quantas-chamadas?)
1
```

**Resposta:**

```
(define (faz-monitor proc)
  (let ((chamadas 0))
    (lambda (arg)
      (cond ((eq? arg 'quantas-chamadas?) chamadas)
            ((eq? arg 'inicializa-contador) (set! chamadas 0))
            (else (set! chamadas (add1 chamadas))
                  (proc arg))))))
```

**Exercício 10.6**

Recorde que um procedimento para calcular números de Fibonacci pode ser definido em Scheme da seguinte forma:

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

Esta versão tem, no entanto, o problema de ser muito pouco eficiente. Use um procedimento com estado interno para escrever uma versão eficiente do procedimento que calcula os números de Fibonacci, que vai guardando os números da sequência de Fibonacci à medida que os vai calculando. O procedimento deve manter um registo dos valores que já calculou. Sempre que tiver que calcular um número da sequência, o procedimento vê se já sabe o seu valor e só vai efectuar novos cálculos caso isso seja mesmo necessário.

**Resposta:**

A *memoization* é uma técnica que permite a um procedimento guardar, numa estrutura local, valores que já foram computados, e pode ter uma grande influência na eficiência de um programa. Um procedimento *memoizado* mantém um registo dos valores previamente calculados juntamente com o valor do argumento que lhes deu origem. Quando é pedido ao procedimento para calcular um novo valor, primeiro ele verifica se ele já está registado e se estiver não precisa de o voltar a calcular. Se o valor ainda não tinha sido calculado, calcula-o e regista-o.

```
(define (memoize proc)
  (define (procura arg assoc-list)
    (cond ((lista-vazia? assoc-list) #f)
          ((eqv? arg (car (primeiro assoc-list))) (cdr (primeiro assoc-list)))
          (else (procura arg (resto assoc-list)))))
  (let ((valores-guardados (nova-lista)))
    (lambda (x)
      (let ((valor-anterior (procura x valores-guardados)))
        (if valor-anterior
            valor-anterior
            (let ((valor-calculado (proc x)))
              (set! valores-guardados (insere (cons x valor-calculado)
  valores-guardados))
              valor-calculado)))))))
```

```
(define memo-fib
  (memoize (lambda (num)
             (cond ((= num 0) 0)
                   ((= num 1) 1)
                   (else (+ (memo-fib (- num 1))
                             (memo-fib (- num 2))))))))
```

```
> (time (fib 5))
cpu time: 0 real time: 0 gc time: 0
5
```

```
> (time (fib 35))
cpu time: 12488 real time: 12516 gc time: 0
9227465
```

```
> (time (fib 36))
cpu time: 20187 real time: 20210 gc time: 0
14930352
```

```
> (time (memo-fib 30))
cpu time: 0 real time: 0 gc time: 0
832040
```

```
> (time (memo-fib 35))
cpu time: 0 real time: 0 gc time: 0
9227465
```

```
> (time (memo-fib 1000))
cpu time: 20 real time: 20 gc time: 0
4346655768693745643568852767504062580256466051737178040248172908953655541794905189040
```

```
> (time (memo-fib 5))
```

```
cpu time: 0 real time: 1 gc time: 0
5
```

### Exercício 10.7

Considere que um cinema com várias salas quer manter o registo do número de espectadores que tem em cada sessão de cada sala, bem como da receita arrecadada em cada sessão.

Usando procedimentos com estado interno, escreva uma versão simplificada do programa para efectuar a gestão das receitas do cinema e mostre como é que ele pode ser utilizado.

Considere que os bilhetes custam 3Euro para crianças e 5Euro para adultos.

### Resposta:

```
(define (faz-sessao sala num-lugares)
  (let ((num-espectadores 0)
        (receita 0))
    (lambda (arg)
      (cond ((eq? arg 'mostra-num-espectadores) num-espectadores)
            ((eq? arg 'mostra-receita) receita)
            ((eq? arg 'mostra-num-lugares) num-lugares)
            ((eq? arg 'mostra-sala) sala)
            ((<= (add1 num-espectadores) num-lugares)
             (set! num-espectadores (add1 num-espectadores))
             (set! receita (+ receita arg)))
            (else (error "esta a tentar exceder a capacidade da sala" sala))))))

(define (entra-sessao sessao pessoa)
  (if (procedure? sessao)
      (cond ((eq? pessoa 'adulto) (sessao 5))
            ((eq? pessoa 'crianca) (sessao 3))
            (else (error "entra-sessao: espera sessao e pessoa, recebeu" pessoa)))
      (error "entra-sessao: espera procedimento no 1o arg, recebeu" sessao)))

> (define sessao1salal1 (faz-sessao 'salal1 40))

> (entra-sessao sessao1salal1 'adulto)

> (entra-sessao sessao1salal1 'adulto)

> (entra-sessao sessao1salal1 'crianca)

> (sessao1salal1 'mostra-sala)
salal1

> (sessao1salal1 'mostra-num-espectadores)
3

> (sessao1salal1 'mostra-receita)
13

> (define sessao2salal1 (faz-sessao 'salal1 30))
```

```
> (entra-sessao sessao2sala1 'adulto)
> (entra-sessao sessao2sala1 'adulto)
> (sessao2sala1 'mostra-sala)
sala1
> (sessao2sala1 'mostra-num-espectadores)
2
> (sessao2sala1 'mostra-receita)
10
> (define sessao1sala3 (faz-sessao 'sala3 2))
> (entra-sessao 'ola 'pois)
entra-sessao: espera procedimento no 1o arg, recebeu ola
> (entra-sessao sessao1sala3 'ola)
entra-sessao: espera sessao e pessoa, recebeu ola
> (entra-sessao sessao1sala3 'adulto)
> (entra-sessao sessao1sala3 'adulto)
> (sessao1sala3 'mostra-num-espectadores)
2
> (entra-sessao sessao1sala3 'crianca)
esta a tentar exceder a capacidade da sala sala3
> (sessao1sala3 'mostra-sala)
sala3
> (sessao1sala3 'mostra-num-espectadores)
2
> (sessao1sala3 'mostra-receita)
10
```

Nota: é necessário usar os procedimentos correctos, senão os resultados não são os esperados.



## 11 Avaliação Baseada em Ambientes

### Sumário:

- Avaliação Baseada em Ambientes

### Resumo:

- O novo modelo de avaliação é necessário por causa da introdução do `set!`, pois agora diferentes chamadas ao mesmo procedimento com o mesmo argumento podem ter resultados diferentes, dependendo do *ambiente* em que forem avaliadas, isto é, do estado do procedimento e da forma como ele evolui ao longo do programa.
- Um *enquadramento* é constituído por um conjunto (possivelmente vazio) de ligações, as quais associam um nome a um valor. Num *enquadramento* não podem existir duas ligações distintas para o mesmo nome. Os *enquadramentos* estão organizados numa estrutura hierárquica, correspondente a uma árvore, em que cada *enquadramento* está associado ao seu *enquadramento* envolvente. O *enquadramento* global é especial e não tem *enquadramento* envolvente.
- O *ambiente* associado a um *enquadramento* é a sequência de *enquadramentos* constituída por esse *enquadramento* e por todos os seus *enquadramentos* envolventes.
- O *valor de uma variável* em relação a um *ambiente* é o valor da ligação dessa variável no primeiro *enquadramento* que contém uma ligação para essa variável. Se não existir nenhuma ligação para a variável, ela diz-se *não-ligada*.
- Cada procedimento está associado a um *objecto procedimental*, que contém informação acerca do código do procedimento (parâmetros e corpo) e a indicação do *enquadramento* em que o procedimento foi criado.
- Para aplicar um procedimento o Scheme cria um *enquadramento* que liga os parâmetros formais do procedimento aos parâmetros concretos que lhe são passados e que tem como *enquadramento* envolvente o *enquadramento* em que o procedimento foi definido.
- Avaliação com base em ambientes
  1. O valor de uma constante é a própria constante.
  2. O valor duma variável num *ambiente* é dado pela ligação da variável no primeiro *enquadramento* contendo essa variável.
  3. A avaliação de uma expressão lambda cria um *objecto procedimental*, contendo o código (corpo e parâmetros) da expressão lambda, e uma ligação ao *enquadramento* em que a expressão lambda é avaliada.
  4. A forma especial `define` cria uma ligação do nome que menciona, ao *objecto computacional* que gera, no *ambiente* em que é avaliada.
  5. Para avaliar a atribuição usando `set!` num *ambiente*, altera-se o valor da ligação da variável indicada. Se a variável não estiver ligada, o `set!` deve assinalar um erro.

6. Para avaliar uma combinação, avaliam-se as subexpressões na combinação (por qualquer ordem) e aplica-se o objecto procedimental correspondente à primeira subexpressão aos valores das restantes subexpressões.
  7. Para aplicar um objecto procedimental a um conjunto de argumentos:
    - (a) Cria-se um novo enquadramento.
    - (b) Este enquadramento é envolvido pelo enquadramento referido no objecto procedimental.
    - (c) No novo enquadramento criam-se as ligações entre os parâmetros formais e os parâmetros concretos.
    - (d) Avalia-se o corpo do procedimento no novo ambiente.
- Resumo:
    - Criam-se enquadramentos quando se chama um procedimento e quando se avalia um `let`. Nesse enquadramento ficam os parâmetros do procedimento ou as variáveis do `let` com os respectivos valores. Aponta para onde apontava o procedimento ou para onde o `let` foi avaliado. Neste enquadramento avalia-se o corpo do procedimento ou do `let`.
    - Criam-se novas entradas num enquadramento quando se avalia um `define`.
    - Alteram-se valores de entradas num enquadramento quando se avalia um `set!`.
    - Criam-se “olhinhos”, que são procedimentos, quando se avalia uma expressão `lambda`. O olho esquerdo aponta para os parâmetros e corpo do procedimento; o olho direito para o enquadramento onde foi definido.

**Exercício 11.1**

Desenhe o diagrama dos ambientes criados pela seguinte interação:

```
> (define x 63)

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

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

> (sum-sq 3 4)
```

**Resposta:**

**Exercício 11.2**

Desenhe o diagrama dos ambientes criados pela seguinte interação:

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

> (define addthree (make-adder 3))

> (define addfive (make-adder 5))

> (addfive 7)

> (addthree 7)
```

**Resposta:**

**Exercício 11.3**

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

Exemplo A:

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

Exemplo B:

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

**Resposta:**

**Exercício 11.4**

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.

**Resposta:**

## 12 O Tipo Vector

### Sumário:

- O tipo *vector*

### Resumo:

- Um *vector* é um tipo estruturado de informação cujos elementos são acedidos indicando a posição do elemento dentro da estrutura.
  - *Constructores*: `make-vector`, `vector`
  - *Selectores*: `vector-ref`, `vector-length`
  - *Modificadores*: `vector-set!`
  - *Reconhedores*: `vector?`
  - *Testes*: `equal?`
- Especificação

```
make-vector : integer -> vector
vector     : universal^k -> vector
vector-ref : vector X integer -> universal
vector-length : vector -> integer
vector-set! : vector X integer X universal -> vector
vector?    : universal -> boolean
equal?     : vector X vector -> boolean
```

**Exercício 12.1**

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 outro vector com os resultados produzidos ao aplicar o procedimento a cada elemento do vector recebido.

```
> (vector-map abs '#(-10 2.5 -11.6 17))
#4(10 2.5 11.6 17)

> (define v '#(1 2 3))

> (vector-map (lambda (x) (* x x)) v)
#3(1 4 9)

> v
#3(1 2 3)
```

**Resposta:**

```
(define (vector-map proc vec)
  (let* ((len (vector-length vec))
         (result (make-vector len)))
    (define (iter pos)
      (cond ((= pos len) result)
            (else (vector-set! result pos (proc (vector-ref vec pos)))
                  (iter (+ pos 1)))))
    (iter 0)))

;;; ou, usando o "do",

(define (vector-map proc vec)
  (let* ((len (vector-length vec))
         (result (make-vector len)))
    (do ((i 0 (+ i 1)))
        ((= i len) result)
      (vector-set! result i (proc (vector-ref vec i)))))
```

**Exercício 12.2**

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

```
> (vector-map-into! abs '#(-10 2.5 -11.6 17))
#4(10 2.5 11.6 17)

> (define v '#(1 2 3))

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

> v
#3(1 4 9)
```

**Resposta:**

```
(define (vector-map-into! proc vec)
  (let ((len (vector-length vec)))
    (define (iter pos)
      (cond ((= pos len) vec)
            (else (vector-set! vec pos (proc (vector-ref vec pos)))
                  (iter (+ pos 1)))))
      (iter 0)))
```

;;; ou, usando o "do",

```
(define (vector-map-into! proc vec)
  (let ((len (vector-length vec)))
    (do ((i 0 (+ i 1)))
        ((= i len) vec)
      (vector-set! vec i (proc (vector-ref vec i)))))
```

**Exercício 12.3**

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

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

```
> (define v '#(1 2 3))
```

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

```
> v
#3(3 2 1)
```

**Resposta:**

```
(define (vector-reverse! vec)
  (define (iter pos1 pos2)
    (if (>= pos1 pos2)
        vec
        (let ((el1 (vector-ref vec pos1)))
          (vector-set! vec pos1 (vector-ref vec pos2))
          (vector-set! vec pos2 el1)
          (iter (+ pos1 1) (- pos2 1)))))
      (iter 0 (- (vector-length vec) 1)))
```

;;; ou, usando o "do",

```
(define (vector-reverse! vec)
  (do ((pos1 0 (+ pos1 1))
      (pos2 (- (vector-length vec) 1) (- pos2 1)))
      ((>= pos1 pos2) vec)
    (let ((el1 (vector-ref vec pos1)))
```

```
(vector-set! vec pos1 (vector-ref vec pos2))
(vector-set! vec pos2 ell))))
```

### Exercício 12.4

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
```

### Resposta:

;;; Este algoritmo corresponde a uma procura sequencial

```
(define (vector-search n vec)
  (let ((len (vector-length vec)))
    (define (iter pos)
      (cond ((= pos len) #f)
            ((= n (vector-ref vec pos)) pos)
            (else (iter (+ pos 1)))))
    (iter 0)))
```

;;; ou, usando o "do",

```
(define (vector-search n vec)
  (do ((len (vector-length vec))
      (i 0 (+ i 1))
      (found? #f (= n (vector-ref vec i))))
      ((or (= i len) found?) (if found? (- i 1) #f))))
```

### Exercício 12.5

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-2`, 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?

**Resposta:**

1. Metade.
2. 

```
(define (vector-search-2 n vec)
  (let ((len (vector-length vec)))
    (define (iter pos)
      (cond ((= pos len) #f)
            ((= n (vector-ref vec pos)) pos)
            ((< n (vector-ref vec pos)) #f)
            (else (iter (+ pos 1)))))
      (iter 0)))
```
3. Metade em ambos os casos.

**Exercício 12.6**

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-procura-binaria` que realiza uma procura binária num vector. Qual a ordem de crescimento deste procedimento?

**Resposta:**

```
(define (vector-procura-binaria n vec)
  (define (iter start end)
    (if (> start end)
        #f
        (let* ((middle (quotient (+ start end) 2))
              (mid-el (vector-ref vec middle)))
          (cond ((= n mid-el) middle)
                ((> n mid-el) (iter (+ middle 1) end))
                (else (iter start (- middle 1)))))))
    (iter 0 (- (vector-length vec) 1)))
```

Este procedimento tem uma ordem de crescimento logarítmica.

**Exercício 12.7**

Defina um procedimento `vector-junta-string` que recebe uma string e um vector e retorna um outro vector com os resultados produzidos ao concatenar a string com cada elemento do vector original.

```
> (define v '#("ontem" "hoje" "amanha"))

> (vector-junta-string "ola " v)
#3("ola ontem" "ola hoje" "ola amanha")

> v
#3("ontem" "hoje" "amanha")
```

**Resposta:**

```
(define (vector-junta-string string vec)
  (let* ((len (vector-length vec))
         (result (make-vector len)))
    (define (iter pos)
      (cond ((= pos len) result)
            (else (vector-set! result pos (string-append string (vector-ref vec pos))
                               (iter (+ pos 1))))))
    (iter 0)))
```

**Exercício 12.8**

Defina um procedimento `vector-ordenado?` que recebe um predicado e um vector e retorna o valor lógico `#t` se o vector estiver ordenado de acordo com esse predicado e `#f` caso contrário.

```
> (vector-ordenado? string<? '#("ontem" "hoje" "amanha"))
#f

> (vector-ordenado? <= '#(1 2 2 3 4))
#t

> (vector-ordenado? > '#())
#t
```

**Resposta:**

```
(define (vector-ordenado? pred vec)
  (let ((len (vector-length vec)))
    (define (iter pos)
      (cond ((= pos len) #t)
            ((pred (vector-ref vec (- pos 1)) (vector-ref vec pos))
             (iter (+ pos 1)))
            (else #f)))
    (if (<= len 1)
        #t
        (iter 1))))
```

## 13 Estruturas Mutáveis: Filas, Pilhas e Estruturas Circulares

### Sumário:

- Estruturas mutáveis: Filas, Pilhas, e Estruturas Circulares.

### Resumo:

- Uma estrutura de informação diz-se mutável quando é modificada destrutivamente à medida que o programa é executado.
- *Modificadores para pares*: `set-car!`, `set-cdr!`

### Filas

- As filas correspondem a uma sequência de elementos com comportamento FIFO.
- As filas representadas como listas têm o problema que não são modificadas destrutivamente à medida que são colocados e/ou retirados elementos.
- Filas com indicação do início e do fim: uma fila corresponde a um par, cujo primeiro elemento contém uma indicação sobre o início da fila e cujo segundo elemento contém uma indicação sobre o fim da fila. Os elementos da fila são representados como uma lista.
- Operações básicas:
  - *Construtores*: `nova-fila`, `coloca!`
  - *Selectores*: `inicio`, `retira!`
  - *Reconhecedores*: `fila?`, `fila-vazia?`
  - *Testes*: `filas=?`

"Tipo Fila com indicação de inicio e fim"

```
(define (nova-fila)
  (cons () ()))

(define (coloca! fila elem)
  (let ((novo (cons elem ())))
    (if (fila-vazia? fila)
        (begin
          (set-car! fila novo)
          (set-cdr! fila novo))
        (let ((fim-fila (cdr fila)))
          (set-cdr! fim-fila novo)
          (set-cdr! fila novo)))
      fila))
```

```

(define (inicio fila)
  (if (fila-vazia? fila)
      (error "inicio: fila nao pode ser vazia")
      (car (car fila))))

; considerando q basta o primeiro ser vazio para a fila ser vazia.
(define (retira! fila)
  (if (fila-vazia? fila)
      (error "retira: fila nao pode ser vazia")
      (begin (set-car! fila (cdr (car fila)))
              fila)))

(define (fila? x)
  (define (ultimo x)
    (if (null? (cdr x))
        x
        (ultimo (cdr x))))
  (cond ((pair? x)
         (if (null? (car x))
             #t
             (and (list? (car x))
                  (eq? (ultimo (car x)) (cdr x))))))
        (else #f)))

(define (fila-vazia? fila)
  (null? (car fila)))

; so usei o elem= para ter a mesma interface do livro
(define (filas=? fila1 fila2 elem=?)
  (equal? (car fila1) (car fila2)))

(define (escreve-fila fila)
  (define (escreve fila)
    (if (not (null? fila))
        (begin
            (display (car fila))
            (if (not (null? (cdr fila)))
                (display " "))
            (escreve (cdr fila))))))
  (display "<")
  (escreve (car fila))
  (display "<")
  (newline))

```

**Pilhas**

- As pilhas correspondem a uma sequência de elementos com comportamento LIFO.
- Operações básicas:
  - *Construtores*: nova-pilha, empurra!
  - *Selectores*: topo, tira!
  - *Reconhecedores*: pilha?, pilha-vazia?
  - *Testes*: pilhas=?
- Uma pilha vazia é representada por uma caixa contendo (). Uma pilha não vazia é representada por uma caixa contendo uma lista com os elementos da pilha, em que o primeiro elemento da lista é o que está no topo da pilha.

```
(define (nova-pilha)
  (box ()))

(define (empurra! elem pilha)
  (set-box! pilha (cons elem (unbox pilha)))
  pilha)

(define (topo pilha)
  (let ((conteudo (unbox pilha)))
    (if (null? conteudo)
        (error "topo: pilha nao tem elementos")
        (car conteudo))))

(define (tira! pilha)
  (let ((conteudo (unbox pilha)))
    (if (null? conteudo)
        (error "tira!: pilha nao tem elementos")
        (begin
         (set-box! pilha (cdr conteudo))
         pilha))))

(define (pilha? x)
  (if (box? x)
      (list? (unbox x))
      #f))

(define (pilha-vazia? pilha)
  (null? (unbox pilha)))

; so usei o elem= para ter a mesma interface do livro
(define (pilhas=? pilha1 pilha2 elem=)
  (equal? (unbox pilha1) (unbox pilha2)))

(define (escreve-pilha pilha)
  (define (escreve-pilha-aux p)
    (cond ((null? p)
           (display "==="))
```

```

        (newline))
      (else (display (car p))
            (newline)
            (escreve-pilha-aux (cdr p))))))
(escreve-pilha-aux (unbox pilha))

```

## Ponteiros

- Um ponteiro é um objecto computacional que *aponta*. Com a utilização de ponteiros, não estamos interessados no valor do ponteiro, mas sim no valor para onde ele aponta.
- `(equal? x1 x2)` tem valor verdadeiro se `x1` e `x2` têm a mesma estrutura e o mesmo conteúdo, independentemente de serem ou não o mesmo objecto computacional.
- `(eq? x1 x2)` tem valor verdadeiro se `x1` e `x2` correspondem ao mesmo objecto computacional, isto é, se partilham a mesma estrutura.

**Exercício 13.1**

Considere definido o tipo *Fila*, com as operações *nova-fila*, *coloca!*, *inicio*, *retira!*, *fila?*, *fila-vazia?*, *filas=?*, com os significados que foram usados no livro.

Escreva um procedimento *procura-largura* que, dados um elemento inicial, uma lista de procedimentos que geram os sucessores de um nó, e um predicado que indica se já se atingiu o objectivo, executa uma procura em largura. Este procedimento deve retornar uma lista com o caminho desde a solução até ao nó inicial.

Usando esse procedimento, escreva um procedimento *seq* que determina o caminho para chegar a um determinado número, começando em 1, e em que os sucessores de um número são o dobro do número e o dobro mais um.

```
> (seq 11)
(11 5 2 1)
```

```
> (seq 1000)
(1000 500 250 125 62 31 15 7 3 1)
```

**Resposta:**

```
(define (procura-largura inicial sucs objectivo?)
  (let ((fila (coloca! (nova-fila) (list inicial))))
    (define (gera-sucessores elem sucs)
      (if (not (null? sucs))
          (begin
            (coloca! fila (cons ((car sucs) (car elem)) elem))
            (gera-sucessores elem (cdr sucs))))))
    (define (procura)
      (let ((no (inicio fila)))
        (retira! fila)
        (if (objectivo? (car no))
            no
            (begin
              (gera-sucessores no sucs)
              (procura))))))
    (procura)))

(define (seq num)
  (procura-largura 1
    (list (lambda (n) (* 2 n))
          (lambda (n) (+ 1 (* 2 n))))
    (lambda (n) (= n num))))
```

**Exercício 13.2**

Considere definido o tipo *Pilha*, com as operações *nova-pilha*, *empurra!*, *topo*, *tira!*, *pilha?*, *pilha-vazia?*, *pilhas=?*, com os significados que foram usados no livro.

Use esse tipo para escrever um procedimento *lista-simetrica?*, que verifica se uma lista de elementos é simétrica relativamente ao elemento central. Este procedimento deve receber como argumentos a lista e o elemento central.

Nota: Assuma que o elemento central aparece na lista uma única vez.

```

> (lista-simetrica? '(1 2 3 ola 3 2 1) 'ola)
#t

> (lista-simetrica? '(pois I tres quatro) 'I)
#f

> (lista-simetrica? '(1 2 3 I 1) 'I)
#f

```

**Resposta:**

```

(define (lista-simetrica? lst central)
  (let ((pilha (nova-pilha)))
    (define (empurrador lst)
      (cond ((null? lst) #f)
            ((eq? (car lst) central)
             (tirador (cdr lst)))
            (else (empurra! (car lst) pilha)
                    (empurrador (cdr lst)))))
    (define (tirador lst)
      (cond ((null? lst) (pilha-vazia? pilha))
            ((pilha-vazia? pilha) #f)
            ((eq? (car lst) (topo pilha))
             (tira! pilha)
             (tirador (cdr lst)))
            (else #f)))
    (if (null? lst)
        #t
        (empurrador lst))))

```