Capítulo 4- Procedimentos como objectos de 1ª classe

Procedimentos com um número não fixo de argumentos
Procedimentos como argumentos
Procedimentos como valores de retorno
Procedimentos como elementos de estruturas de dados

Exercícios e exemplos
Projecto - CODEC

Os objectos de 1ª classe são, normalmente, os operandos (números, booleanos, pares e símbolos), entidades que podem ser processadas e, por isso, reconhecidos como objectos *passivos*. Por seu turno, os procedimentos são objectos *activos*, pois associam-se-lhes actividades de processamento. As propriedades dos objectos de 1ª classe são a seguir indicadas, com exemplos aplicados aos números:

1. Ligam-se a símbolos, `(define a 35)`;
2. São passados como argumentos de procedimentos, `(quadrado 5)`;
3. São devolvidos como valores de retorno de procedimentos, `(+ 3 (quadrado 5))`;
4. São elementos de estruturas compostas de dados, `(list 1 2 3)`.

Poderá parecer estranho, mas em *Scheme*, os procedimentos são também considerados objectos de 1ª classe. Em *Scheme*, todos os objectos têm estatuto de 1ª classe, sejam eles números, booleanos, pares e símbolos, não sendo excepção os procedimentos primitivos, disponibilizados pela linguagem, e os procedimentos compostos, definidos pelo programador.
A propriedade 1. dos objectos de 1ª classe tem sido amplamente demonstrada para os procedimentos. Estes, através de *lambda*, ligam-se aos símbolos escolhidos para nome dos procedimentos. As restantes três propriedades serão verificadas no decorrer do presente capítulo. O facto dos procedimentos poderem ser tomados como argumentos ou até mesmo como valores de retorno de outros procedimentos vai permitir o desenvolvimento de soluções gerais e muito flexíveis para certos padrões de computação, como teremos oportunidade de verificar.
O capítulo inicia-se com a apresentação de procedimentos que funcionam com um número não fixo de argumentos, caso não considerado nos capítulos anteriores. Seguem-se os *procedimentos de ordem mais elevada*, assim designados por admitirem como parâmetros outros procedimentos, e não só os números, booleanos, pares e símbolos, como acontece nos *procedimentos de 1ª ordem*. Serão também analisados os procedimentos que apresentam, como valor de retorno, outros procedimentos, característica eventualmente perturbadora, mas que potencia outras vias de programação muito interessantes, algumas demonstradas neste capítulo, outras mais tarde, no capítulo de introdução à *programação orientada por objectos*.

1- Procedimentos com um número não fixo de argumentos

Os procedimentos desenvolvidos até aqui apresentavam uma característica comum, o número de argumentos utilizados nas respectivas chamadas coincidia sempre com o número de parâmetros especificados, entre parêntesis, a seguir ao *lambda*. Em todos estes casos, o número

de argumentos ficava estabelecido, quando se definiam os parâmetros dos procedimentos. Mas não terá de ser sempre assim. O próprio Scheme disponibiliza procedimentos que não seguem esta regra, pois aceitam um número não fixo de argumentos. São disto exemplo: +, -, *, /, <, e outros.

```
↦(+ 1 2)                          ; chamada com 2 argumentos
3
↦(+ 1 2 3)                        ; chamada com 3 argumentos
6
```

Os procedimentos com um número *fixo* de argumentos são definidos colocando uma lista de parâmetros a seguir a *lambda*.

```
(define proc-com-numero-fixo-de-argumentos
   (lambda (param-1 param-2 param-3 ...)

       corpo-do-procedimento))
```

Os procedimentos com um número *não fixo* de argumentos são definidos de uma forma diferente. Neste caso, imediatamente a seguir a *lambda*, encontra-se não uma lista com um certo número de parâmetros, mas apenas um símbolo. Foi utilizado o símbolo *argumentos*, mas outro qualquer serviria.

```
(define proc-com-numero-nao-fixo-de-argumentos
   (lambda argumentos

       corpo-do-procedimento))
```

Depois da chamada

```
(proc-com-numero-nao-fixo-de-argumentos arg1 arg2  ...)
```

o símbolo *argumentos* fica ligado à lista (arg1 arg2 ...), constituída por todos o argumentos utilizados, cujo número poderá variar em cada chamada.

Exemplo 4.1
O procedimento *soma-primeiro-e-ultimo* devolve a soma do primeiro e do último argumentos que lhe são passados em cada chamada.

```
↦(soma-primeiro-e-ultimo 1 2)           ; chamada com 2 argumentos
3
↦(soma-primeiro-e-ultimo 1 2 3 4 5 6)   ; chamada com 6 argumentos
7
↦(soma-primeiro-e-ultimo 1)             ; chamada com 1 argumento
erro. Incorrecto numero de argumentos
↦(soma-primeiro-e-ultimo)               ; chamada com zero argumentos
erro. Incorrecto numero de argumentos
```

Na definição do procedimento, o símbolo *argumentos* é tratado como uma lista, de acordo com o que foi dito.

```
(define soma-primeiro-e-ultimo
  (lambda argumentos

    (let ((comprimento (length argumentos)))
      (if(< comprimento 2)
         (display "erro. Incorrecto numero de argumentos")
         (+ (car argumentos)
            (list-ref argumentos
                      (sub1 comprimento)))))))
```

Segue-se uma versão alternativa de *soma-primeiro-e-ultimo*, que não utiliza o procedimento primitivo *list-ref*. Nesta nova versão foi definido um procedimento interno, *procura-ultimo*, que recebe, garantidamente, uma lista de, pelo menos, 2 elementos. O seu parâmetro, *lis*, é ligado à lista *argumentos*, que lhe é passada na chamada `(procura-ultimo argumentos)`.

```
(define soma-primeiro-e-ultimo
  (lambda argumentos

    (letrec ((procura-ultimo              ; este procedimento interno
               (lambda (lis)              ; considera que lis tem sempre,
                 (if (null? (cdr lis))    ; pelo menos, 2 elementos
                     (car lis)
                     (procura-ultimo (cdr lis))))))

      (cond
        ((< (length argumentos) 2)
         (display "erro. Incorrecto numero de argumentos"))
        (else
          (+ (car argumentos)
             (procura-ultimo argumentos)))))))
```

Exercício 4.1
Procurar outra alternativa para *soma-primeiro-e-ultimo* que se baseia no procedimento *reverse* e não recorre a qualquer procedimento auxiliar.

2- Procedimentos de ordem mais elevada

Os procedimentos que não aceitam outros procedimentos como argumentos são designados por *procedimentos de 1ª ordem*. A partir de agora, também serão considerados procedimentos que aceitam outros procedimentos como argumentos e que, por esse facto, se designam por *procedimentos de ordem mais elevada*. Como exemplos, apontam-se alguns procedimentos primitivos do *Scheme*, *map*, *for-each* e *apply*, e outros criados para resolução de situações enunciadas no decorrer do capítulo.

A chamada do procedimento primitivo *map* apresenta o seguinte aspecto:

```
(map proc lista1 lista2 ...)
```

e devolve a lista cujo primeiro elemento é obtido aplicando *proc* ao primeiro elemento de todas as listas que são argumentos, o segundo elemento é obtido aplicando *proc* ao segundo elemento de todas as listas, e assim sucessivamente até ao último elemento das listas, que deverão ter

todas o mesmo comprimento[1]. O procedimento *proc* deverá aceitar um número de argumentos igual ao número de listas que aparecem em cada chamada.

```
↦(map max '(1 2 3 4) '(4 3 2 1))        ; o procedimento max aceita qualquer
(4 3 3 4)                                ; número de argumentos, neste caso, 2 em
                                         ; cada chamada

↦(map add1 '(1 2 3 4))                    ; o procedimento add1 só aceita
(2 3 4 5)                                 ; um argumento. Neste caso, a
                                          ; chamada só especifica uma lista

↦(map (lambda (x) (* x x)) '(1 2 3))     ; idêntico ao caso anterior
(1 4 9)
```

Em certas situações, a utilização de *map* não é a solução mais conveniente. Por exemplo, quando não se espera a lista resultante, como acontecia nos 3 exemplos indicados, mas apenas interessam os efeitos laterais de um procedimento. É o que acontece quando se aplica *display* aos vários elementos de uma lista. Não esperamos uma lista como resultado, mas os efeitos laterais de *display*, utilizando-se para tal o procedimento primitivo *for-each*. A chamada de *for-each* apresenta a forma:

```
(for-each proc lista1 lista2 ...)
```

e, apenas para efeitos laterais, aplica *proc* ao primeiro elemento de todas as listas que são argumentos, depois ao segundo elemento de todas as listas, e assim sucessivamente até ao último elemento das listas, que deverão ter todas o mesmo comprimento. O procedimento *proc* deverá aceitar um número de argumentos igual ao número de listas que surgem em cada chamada.

Exemplo 4.2

Para ilustrar a funcionalidade de *for-each*, vamos definir um procedimento que escreve uma linha de texto e responde da seguinte maneira:

```
↦(escreve-linha "x = " 5 "; y = " 6)        ; chamada com 4 argumentos
x = 5; y = 6

↦(escreve-linha "Ola'." " " "Estas bom?")   ; chamada com 3 argumentos
Ola'. Estas bom?
```

No primeiro destes exemplos, quatro chamadas a *display* e uma a *newline* seriam necessárias para produzir o efeito apresentado. Com o procedimento *escreve-linha*, este tipo de situação simplifica-se.

```
(define escreve-linha
  (lambda args
    (for-each display args)
    (newline)))
```

A chamada do procedimento primitivo *apply* apresenta a seguinte forma:

```
(apply proc lista)
```

e devolve um resultado equivalente a uma chamada de *proc* em que os argumentos seriam todos os elementos da *lista*[2]. Como nada se diz sobre o comprimento da lista, o procedimento *proc* deverá aceitar um número não fixo de argumentos.

---

[1] *map* do EdScheme funciona com listas de comprimento diferente e devolve, como resultado, uma lista cujo comprimento é igual ao comprimento da lista de menor comprimento: `↦(map max '(1 5 3) '(3 2)) devolve (3 5)`

[2] Convém entender perfeitamente qual é a diferença entre dizer que os *argumentos são todos os elementos de uma lista* ou o *argumento é a própria lista*. Os exemplos que se seguem ajudam a esclarecer esta questão

```
↦(apply + '(1 2 3))                    ; equivalente a (+ 1 2 3)
6
↦(apply add1 '(1 2 3))                 ; equivalente a (add1 1 2 3)

add1: wrong number of arguments
↦(apply max '(1 2 3 9 4))[3]                    ; analisar
9                                                ; com
↦(max '(1 2 3 9 4))                              ; muita
                                                   ; atenção
max: wrong argument type pair (expected number)    ; estes três
↦(max 1 2 3 9 4)                                    ; comandos
9
```

Cada um destes três procedimentos primitivos evidencia um certo padrão de computação. Por exemplo, *apply* aplica uma função a todos os membros de uma lista. Trata-se de uma computação genérica, independente de uma função específica. Sem *apply*, seria necessário um procedimento diferente para cada função, apesar de todos eles apresentarem um padrão semelhante. Estas situações ocorrem com frequência e o programador deverá decidir, em cada momento, se é preferível desenvolver uma solução genérica, através de um *procedimento de ordem mais elevada* ou então uma solução específica.

Exemplo 4.3

Podemos imaginar o procedimento *max-fun* no contexto de uma destas situações, em que é necessário determinar o máximo de uma função, num domínio definido por dois limites. Optou-se por uma solução geral, em vez de resolver o problema para uma função específica. O procedimento *max-fun* tem como parâmetros uma função de uma variável, dois valores que definem os limites inferior e superior de um domínio em que a função é contínua e um valor incremental, com o qual se definem os pequenos saltos constantes, realizados para percorrer a função no citado domínio, desde o limite inferior até ao limite superior. O procedimento responde da seguinte maneira:

```
↦(max-fun cos 0 1 .01)
1.0
↦(max-fun sin 0 .5 .01)
0.479425538604203
↦(max-fun (lambda (x) (- (* x x) x)) 0 10 .01)
90.0
```

Na definição de *max-fun* é utilizado *aux*, um procedimento interno recursivo que, garantidamente, recebe os limites que definem o domínio de estudo da função, em ordem crescente. Este cuidado é tido em conta, pois *max-fun* poderá receber aqueles limites em ordem decrescente. Por seu turno, *aux* utiliza o procedimento primitivo *max*.

```
(define max-fun
  (lambda (fun lim-inf lim-sup dx)

    (letrec ((aux
               (lambda (inf sup)
                 (cond
                   ((> inf sup) (f sup))
                   (else
                     (max (f inf) (aux (+ inf dx) sup)))))))

      (if (> lim-inf lim-sup)
          (aux lim-sup lim-inf)
          (aux lim-inf lim-sup)))))
```

[3] O procedimento primitivo *max* aceita um número não fixo de argumentos e devolve o maior deles (Anexo A)

Exercício 4.2

Verificar que o processo recursivo gerado por *max-fun* esgota a memória disponível, com alguma facilidade.

```
↦(max-fun sin 0 1.5 .001)
let: Out of stack space

↦(max-fun sin .5 1.5 .001)
0.997494986604054
```

Definir uma solução iterativa, *O(1)* em termos de espaço, para evitar problemas como o indicado.

Exemplo 4.4

Os procedimentos *soma-inteiros* e *soma-quadrados* apresentam padrões de computação muito semelhantes. Têm como parâmetros dois limites e ambos somam, entre esses dois limites, os inteiros ou os quadrados desses inteiros.

```
(define soma-inteiros
  (lambda (a b)
    (if (> a b)
        0
        (+ a
           (soma-inteiros (add1 a) b)))))

(define soma-quadrados
  (lambda (a b)
    (if (> a b)
        0
        (+ (* a a)
           (soma-quadrados (add1 a) b)))))
```

Estes dois procedimentos podem ser vistos com um somatório dos valores de uma função, calculadas entre dois limites, com passo unitário. No primeiro caso, a função seria f(x) = x, no segundo, f(x) = x * x. Se assim for, os dois procedimentos podem ser considerados casos particulares de um procedimento mais geral, *soma*, com 4 parâmetros: Os dois limites que já apareciam em *soma-inteiros* e *soma-quadrados*, a função que calcula os termos a somar e a função que define o passo.

```
(define soma
  (lambda (fun inf sup passo)
    (if (> inf sup)
        0
        (+ (fun inf)
           (soma fun (passo inf) sup passo)))))
```

Baseados em *soma*, aqueles procedimentos podem agora tomar a seguinte forma:

```
(define soma-intereiros-baseado-em-soma
  (lambda (inf sup)
    (soma (lambda(x) x) inf sup add1)))

(define soma-quadrados-baseado-em-soma
  (lambda (inf sup)
    (soma (lambda(x) (* x x)) inf sup add1)))
```

Exercício 4.3
Por exemplo, a soma dos *n* primeiros ímpares é dada por 1 + 3 + 5 + ... + (2 * n - 1). Utilizando *soma*, escrever em *Scheme soma-impares-baseado-em-soma*, com os parâmetros *inf* e *sup*, que definem, respectivamente, os limites inferior e superior dos ímpares a somar.
Pista:

- Função para cálculo dos termos a somar, $f(x)=2*x-1$
- Função que define o passo, $f(x)=x+2$

Exercício 4.5
O procedimento *pi-v1*, baseado em *soma*, inspira-se na fórmula:

$$\frac{p}{8}=\frac{1}{1*3}+\frac{1}{5*7}+\frac{1}{9*11}+...$$

Definir *pi-v1*, sabendo que tem apenas um parâmetro, *n*, e devolve o valor de *pi*, aproximado aos primeiros *n* termos da fórmula.
Pista:

- Função para cálculo dos termos a somar, $f(x)=\frac{1}{x*(x+2)}$
- Função que define o passo, $f(x)=x+4$

Exercício 4.6
O procedimento *soma* é o caso mais simples de várias abstracções semelhantes. Definir algo de equivalente, mas para o caso do produto, sendo o respectivo procedimento designado por *produto*.

Exercício 4.7
O procedimento *pi-v2*, baseado em *produto*, inspira-se na fórmula:

$$\frac{p}{4}=\frac{2*4*4*6*6*8*...}{3*3*5*5*7*7*...}$$

Escrever *pi-v2* em *Scheme*, sabendo que tem apenas o parâmetro *n*, e devolve o valor de *pi*, aproximado aos primeiros *n* termos da fórmula.
Pista: Começar por identificar

- A função que calcula os termos;
- A função que define o passo.

Por seu turno, os procedimentos *soma* e *produto* podem ser considerados como casos particulares de um procedimento ainda mais geral, *de ordem mais elevada*, designado por *acumulado*. Este procedimento combina uma colecção de termos, através de uma chamada do tipo:

```
(acumulado combinador valor-neutro fun inf sup passo)
```

Os parâmetros *fun*, *inf*, *sup* e *passo* são equivalentes aos parâmetros respectivos de *soma* e *produto*. O parâmetro *valor-neutro* representa o valor neutro da função *fun (por exemplo, 0 em soma e 1 em produto)*. O *combinador* é um procedimento de dois parâmetros que combina o termo corrente com o acumulado dos termos anteriores.

Exercício 4.8
Indicar como se expressam *soma* e *produto* recorrendo a *acumulado*.

Exercício 4.9

Escrever em *Scheme* os procedimentos *acumulado-rec* e *acumulado-iter*, respectivamente, soluções recursiva e iteractiva de *acumulado*.

3- Procedimentos como valores de retorno

Quando se coloca a questão: Qual é a derivada de x^3? A resposta correcta deverá ser $3x^2$. Ou seja, neste caso, a derivada de uma função de x continua a ser uma função de x. E se a derivada da função for calculada através de um procedimento, então não restam dúvidas que este procedimento terá de devolver uma função. No desenvolvimento do procedimento *derivada* utiliza-se a seguinte definição, que supõe *dx* um valor muito pequeno:

$$Df(x) = \frac{f(x+dx) - f(x)}{dx}$$

```
(define derivada
  (lambda (fun dx)

    (lambda (x)
      (/ (- (fun (+ x dx)) (fun x))
         dx))))
```

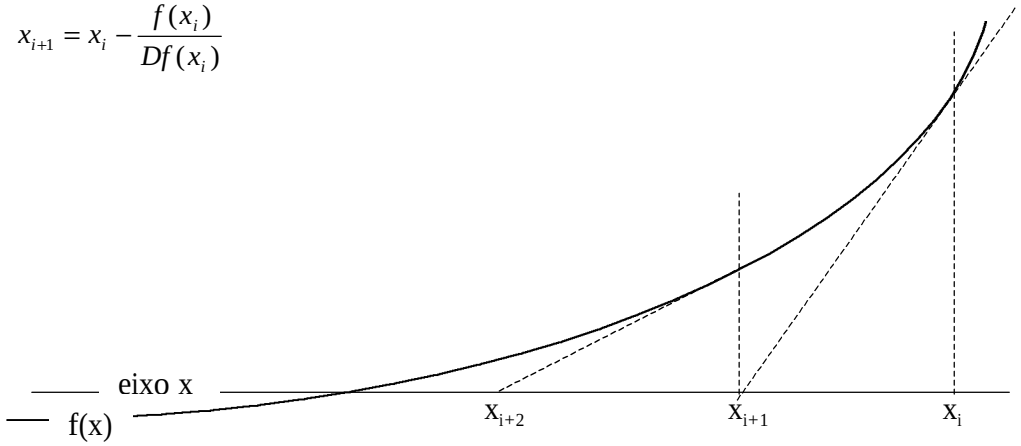
Alguns momentos de reflexão deverão ser tidos neste momento. Pela primeira vez, na definição de um procedimento, aparecem dois *lambdas* seguidos. O que é que isto significa? Apenas significa que o procedimento *derivada*, com dois parâmetros *fun* e *dx*, devolve não um número, um booleano, um símbolo ou um par, mas sim um procedimento, neste caso, um procedimento com o parâmetro *x*.
Verificar, nas chamadas que se seguem, dois níveis de parêntesis. Do nível interno, relacionado com uma chamada a *derivada*, portanto com 2 argumentos, resulta um procedimento. O nível externo é uma chamada ao procedimento que resulta da chamada do nível interno, portanto exibindo apenas um argumento.

```
↦((derivada cos .001) 0)
-0.000499999958325503
↦((derivada (lambda(x)(* 2 x x)) .001) 2)
8.0019999999994
↦((derivada (lambda(x)(* 2 x x)) .001) 3)
12.001999999999
```

Exemplo 4.5

Segundo o *método de Newton-Raphson*, para encontrar as raízes de uma função diferenciável *f(x)*, se x_i é uma aproximação para uma raiz de *f(x)*, uma aproximação melhor será x_{i+1}, em que

$$x_{i+1} = x_i - \frac{f(x_i)}{Df(x_i)}$$



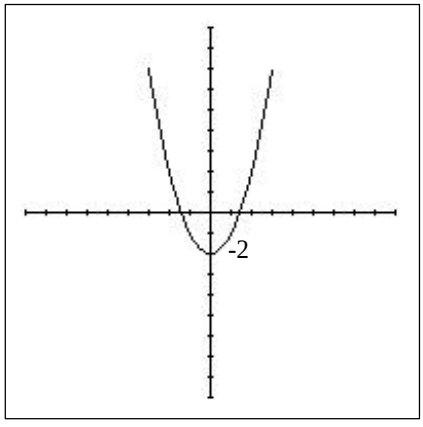
Na figura, é esboçado o *método de Newton-Raphson*. Desenha-se a tangente a *f(x)*, em x_i e verifica-se que a tangente intersecta o eixo dos *xx* em x_{i+1}, mais perto da raíz que x_i. Se, todavia, x_{i+1} não está suficientemente perto da raíz, desenha-se a tangente a *f(x)*, mas agora em x_{i+1} como se fez para x_i.
A ideia a explorar é tentar saber se uma aproximação já é suficientemente boa para ser uma raiz de *f(x)* e, se não for, procurar outra melhor. Assim, o procedimento *newton-raphson* baseia-se em três procedimentos internos, respectivamente, *boa-aprox?*, predicado que indica se a aproximação já é boa, *melhora* que, a partir de uma aproximação, devolve uma aproximação melhor e *aux* um procedimento recursivo que vai requerendo uma melhor aproximação até que esta seja suficientemente boa. Ainda internamente, é ligado ao símbolo *derivada-fun* o resultado de uma chamada ao procedimento *derivada*, o qual devolve, como se sabe, um procedimento de um parâmetro.

```
(define newton-raphson
  (lambda (func aprox)

    (let ((derivada-fun (derivada func 0.001)))

      (let ((melhora
              (lambda (ap)                ; ap é uma aproximação a uma raiz
                (- ap (/ (func ap)        ; de f, que se pretende melhorar
                         (derivada-fun ap)))))

            (boa-aprox?                   ; para verificar se a aproximação é
              (lambda (ap)                ; suficientemente boa...
                (< (abs (func ap)) .00001))))

        (letrec ((aux
                   (lambda (ap)
                     (if (boa-aprox? ap)
                         ap
                         (aux (melhora ap))))))

          (aux aprox))))))
```

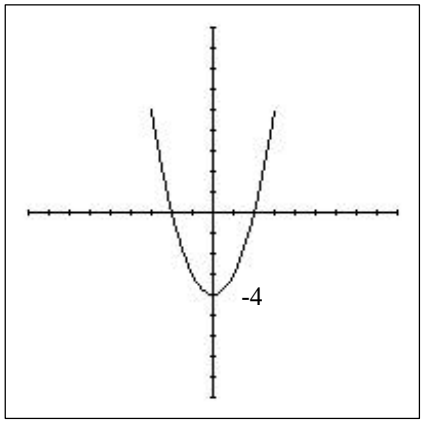
```
↦(newton-raphson (lambda (x) (+ (* x x) (* -5 x) 6)) 1)
2.00000001073126                          ; de facto, x² - 5x + 6 = 0, para x =2
↦(newton-raphson (lambda (x) (+ (* x x x) (* -5 x) 6)) 1)
-2.68909532075017                         ; x³ - 5x + 6 = 0, para x = -2.6890...
```

Seguem-se algumas experiências de *newton-raphson* com equações que definem parábolas.

parabola2
*x * x - 2*

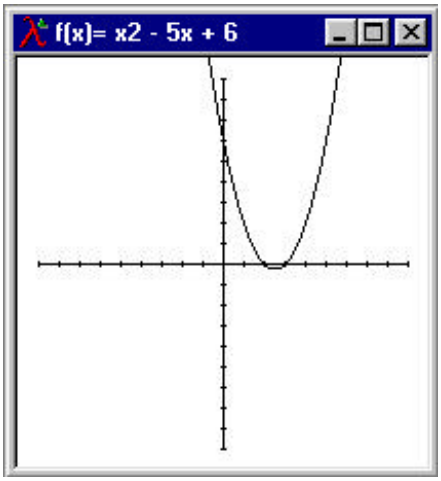


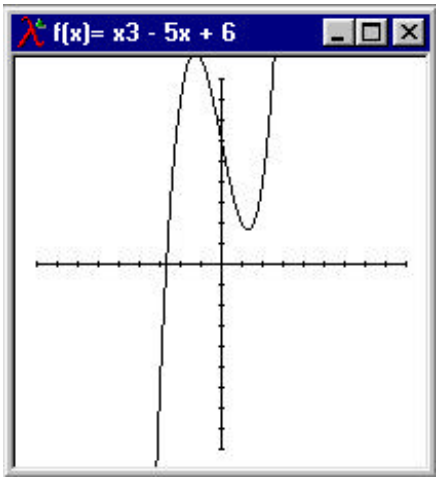
parabola4
*x * x - 4*



```
↦(define parabola2 (lambda (x) (- (* x x) 2)))
parabola2
↦(define parabola4 (lambda (x) (- (* x x) 4)))
parabola4
↦(newton-raphson parabola2 5)
1.41421368307536
↦(newton-raphson parabola2 -5)
-1.41421349682207
↦(newton-raphson parabola4 5)
2.00000000161646
↦(newton-raphson parabola4 -50)
-1.99999997081133
↦(newton-raphson parabola4 -5)
-1.99999999901409
```

Nos exemplos das parábolas foram calculadas duas raízes, para cada uma das funções, pois sabia-se de antemão, pelos desenhos das mesmas, que elas passavam duas vezes pelo eixo dos *xx*. A presença das figuras foi fundamental para dar a conhecer não só a existência das duas raízes, mas também os respectivos valores aproximados. Foi assim possível escolher as aproximações às raízes que viriam a ser utilizadas nas chamadas do procedimento *newton-rapson*. Já para as funções $f(x) = x^2 - 5x + 6$ e $f(x) = x^3 - 5x + 6$, apenas foi encontrada uma raíz para cada uma delas. Será mesmo assim?





Pela análise das figuras conclui-se que, em relação à primeira das funções, uma raíz ficou por encontrar, pois existem duas, uma por volta do valor 2, que foi encontrada, e outra por volta do valor 3. De facto, utilizando, por exemplo, a aproximação 5, em vez de 1, obtém-se a outra raíz.

```
↦(newton-raphson (lambda (x) (+ (* x x) (* -5 x) 6)) 5)
3.00000404844853
```

Exercício 4.10

No contexto do que acaba de ser apresentado, escrever o procedimento *desenha-fun* que desenha uma função *y = f(x)* na janela corrente, com os parâmetros *f*, *origem*, *escala*, *inf*, e *sup*.

⇒ *f* - função a desenhar;

⇒ *origem* - origem de um sistema de eixos, desenhado pelo procedimento *eixos* (ver capítulo anterior) na janela corrente;

⇒ *escala* - escala utilizada pelo procedimento *eixos*, no desenho dos eixos;

⇒ *inf* e *sup* - limites inferior e superior do domínio em que se pretende desenhar a função.

As duas figuras anteriores foram obtidas com a seguinte sequência de comandos:

```
↦(janela 200 200 "f(x)= x3 - 5x + 6")
#[graphics window 7349048]
↦(eixos '(100 100) '(10 10) '(9 9) '(9 9))
()
↦(desenha-fun (lambda (x) (+ (* x x) (* -5 x) 6))
              '(100 100) '(10 10) -10 10)
ok

↦(janela 200 200 "f(x)= x3 - 5x + 6")
#[graphics window 7369064]
↦(eixos '(100 100) '(10 10) '(9 9) '(9 9))
 ()
↦(desenha-fun (lambda (x) (+ (* x x x) (* -5 x) 6))
              '(100 100) '(10 10) -10 10)
ok
```

Na chamada que se segue, é determinado o valor *x* para o qual $x - (x^2 - 5x + 6) = 0$, ou seja, $x = x^2 - 5x + 6$. Este valor de *x* é o chamado *ponto-fixo*[4] de $x^2 - 5x + 6$. O *ponto-fixo* de uma função poderá ser determinado através de um procedimento apropriado, recorrendo a *newton-raphson*.

```
↦(newton-raphson (lambda (x) (- x (+ (* x x) (* -5 x) 6))) 1)
1.26794921520565          ; ponto fixo... x - (x² -5x + 6) = 0 ou x = (x² -5x + 6)
```

```
(define ponto-fixo
  (lambda (f)
    (newton-raphson (lambda(x)(- x (f x))) 1)))
```

```
↦(ponto-fixo (lambda (x)  (+ (* x x) (* -5 x) 6)))
1.26794921520565       ; resultado esperado...
↦(ponto-fixo (lambda (x) (* x x)))
1
```

Exercício 4.11

Com o procedimento *desenha-fun*, desenvolvido no exercício anterior, verificar se as funções utilizadas na ilustração do procedimento *ponto-fixo* têm mais do que um ponto fixo. Definir e desenvolver uma versão melhorada de *ponto-fixo* que permita, numa única chamada, a determinação de todos os pontos fixos de uma função.
Pista: Considerar uma solução que passe pela visualização prévia da função com *desenha-fun*.

Analisemos as chamadas de *newton-Raphson* que se seguem. Na primeira, determina-se o zero da função $9 - x^2 = 0$, equivalente a $x = \sqrt{9}$, ou seja, o zero da função coincide com a raiz quadrada de 9.

[4] O ponto-fixo de uma função é o ponto em que essa função e o seu argumento exibem o mesmo valor.

```
↦(newton-raphson (lambda (x) (- 9  (* x x))) 1)
3.00000001742272
```

Na segunda, determina-se o valor de *x* para o qual $9-x^3=0$, equivalente a $x=\sqrt[3]{9}$. Assim, o zero da função coincide com a raiz cúbica de 9.

```
↦(newton-raphson (lambda (x) (- 9  (* x x x))) 1)
2.08008383433185
```

Finalmente, na terceira, o zero da função coincide com a raiz cúbica de 8.

```
↦(newton-raphson (lambda (x)(- (* x x x) 8)) 1)
2.00000657276036
```

Destes exemplos deduz-se uma ideia para a escrita de procedimentos que calculam a raíz quadrada e a raíz cúbica, como casos particulares do procedimento *newton-raphson*.

```
(define raiz-quadrada
  (lambda (numero)
    (newton-raphson (lambda(x)(- (* x x ) numero)) 1)))

(define raiz-cubica
  (lambda (numero)
    (newton-raphson (lambda(x)(- (* x x x ) numero)) 1)))
```

```
↦(raiz-quadrada 48)
6.92820343812542
↦(sqrt 48)   ; o mesmo cálculo recorrendo directamente a um procedimento do Scheme
6.92820323027551
```

4- Procedimentos como elementos de estruturas de dados

Para ilustrar esta característica dos procedimentos, vamos definir uma lista de procedimentos.

```
(define lista-de-proc (list (lambda(x)(* x x x)) expt car))
```

Analisar com atenção os exemplos que se seguem, os quais demonstram que os procedimentos podem ser elementos de estruturas de dados compostas, neste caso, elementos de uma lista.

```
↦((car lista-de-proc) 5)
125
↦((cadr lista-de-proc) 2 5)
32
↦((caddr lista-de-proc) lista-de-proc)
#[compound 10789976]
↦((caddr lista-de-proc) '((1 2) (3 4)))
(1 2)
```

Exercícios e exemplos de final de capítulo

Segue-se um conjunto de exercícios e exemplos para consolidação da matéria apresentada no capítulo.

Exemplo 4.6

O procedimento primitivo *max* aceita um número não fixo de argumentos e devolve o valor do maior argumento. Escrever uma versão pessoal de *max*, designada por *max-pessoal*.

Solução

```
(define max-pessoal
  (lambda args

    (letrec ((max-aux
               (lambda (lis)
                 (cond
                   ((null? (cdr lis))
                    (car lis))
                   (else
                     (let
                       ((primeiro (car lis))
                        (outros (max-aux (cdr lis))))
                       (if (< primeiro outros)
                           outros
                           primeiro)))))))
      (if (null? args)
          (display "erro. Numero incorrecto de argumentos")
          (max-aux args)))))
```

Exercício 4.12

Seguir, passo a passo, o funcionamento de *max-pessoal*, para a chamada (max-pessoal 2 7 1 3).

Exercício 4.13

Definir o procedimento *pi-v3*, baseado em *soma*, partindo da fórmula:

$$\frac{p}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-...$$

Definir *pi-v3*, sabendo que tem apenas um parâmetro, *n*, e devolve o valor de *pi*, aproximado aos primeiros 2**n* termos da fórmula.

Exercício 4.14

O valor de *e* pode ser aproximado através da fórmula:

$$e=1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+$$

Definir o procedimento *e-aprox*, com o parâmetro *n*, e que devolve o valor de *e* aproximado aos primeiros *n* termos da fórmula. Basear a solução apresentada em *soma* (para somar os *n* termos) e *produto* (para o factorial).

Exercício 4.15

O *golden ratio* pode ser definido pela fórmula:

$$\Phi=\frac{1+\sqrt{5}}{2}\approx 1.61803$$

e pode ser calculado através de:

$$\Phi=1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+}}}}$$

Definir o procedimento *golden-aprox*, com o parâmetro *n*, que devolve o valor de *golden ratio* aproximado aos primeiros *n* termos da fórmula. Basear a solução apresentada em *acumulado*.

Exercício 4.16

A fórmula que se apresenta é uma aproximação do integral de uma função *f(x)*, entre *a* e *b*. Tentar verificar, graficamente, que a aproximação é tanto melhor quanto menor for *dx*.

$$\int_a^b f(x) = \left[f(a+\frac{dx}{2}) + f(a+dx+\frac{dx}{2}) + f(a+2dx+\frac{dx}{2}) + ... \right] dx$$

Definir o procedimento *integral*, baseado em *soma*, partindo da fórmula apresentada.
Pista: Começar por identificar

- A função que calcula os termos a somar;
- A função que define o passo.

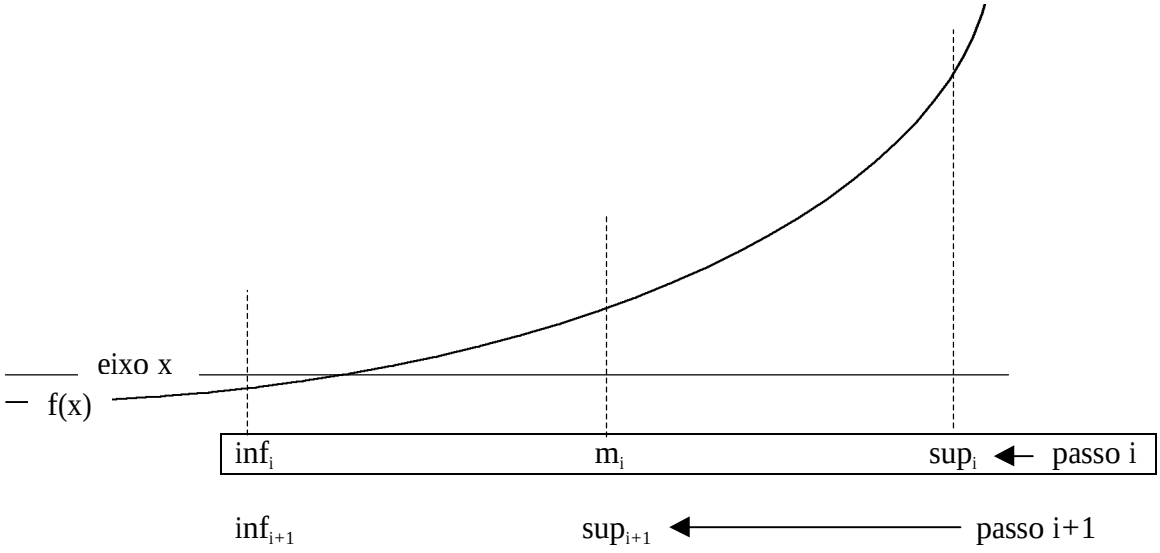
Exercício 4.17

Mais geral que o procedimento *acumulado* (ver Exercício 4.9) é *acumulado-com-filtro*, com os mesmos parâmetros que o seu antecessor e ainda mais um parâmetro adicional, um predicado, que funciona como filtro. Definir o procedimento *acumulado-com-filtro* e, baseado nele, indicar como expressaria a soma dos quadrados dos números primos no intervalo definido pelos limites *inf* e *sup*.
Pista: Numa primeira fase, supor que existe o predicado *primo?*. Posteriormente, escrever também este predicado.

Exercício 4.18

O método da bissecção, para encontrar as raízes de uma função, é esquematizado na figura. No *passo i* do processo de procura, a raíz encontra-se no domínio que tem por limite inferior $x = inf_i$ e por limite superior $x = sup_i$, pois sabe-se que $f(inf_i) < 0 < f(sup_i)$. É, então, determinado o ponto médio do referido limite, m_i, e feitas as seguintes verificações sequencialmente:

⇒ Se $f(m_i)$ é suficientemente próximo de zero, a raíz é tomada como sendo m_i;
⇒ Se $f(m_i) > 0$, a procura continua, no *passo i+1*, com $inf_{i+1} = inf_i$ e $sup_{i+1} = m_i$;
⇒ Se $f(m_i) < 0$, a procura continua, no *passo i+1*, com $inf_{i+1} = m_i$ e $sup_{i+1} = sup_i$.



Escrever em *Scheme* o procedimento *bisseccao* que implementa o método acabado de apresentar, com os parâmetros *fun*, *inf* e *sup*, em que *fun* representa uma função *f(x)*, e *inf* e *sup* definem os limites do domínio em que se procura a raíz, havendo a garantia que *f(inf) < 0 < f(sup)*.

Exemplo 4.7
Definir o procedimento *compoe* que tem dois procedimentos como parâmetros, *p1* e *p2*, e devolve um procedimento de um só parâmetro, *x*. O procedimento resultante aplica *p1* a *p2(x)*, ou seja, `p1(p2(x))`.

Solução

```
(define compoe
 (lambda (f g)

    (lambda (x)
       (f (g x)))))
```

```
↦(define soma-1-e-eleva-a-2 (compoe (lambda(x)(* x x)) add1))
soma-1-e-eleva-a-2
↦(soma-1-e-eleva-a-2 9)
100
```

Exercício 4.19
Definir o procedimento *compoe3* que tem três procedimentos como parâmetros, *p1*, *p2*, e *p3*, e devolve um procedimento de um único parâmetro, *x*. O procedimento resultante faz `p1(p2(p3(x)))`.

Exercício 4.20
Definir o procedimento *compoe-muitas*, que responde da seguinte maneira:

```
↦((compoe-muitas add1 add1 add1 add1) 3)
7
```

Pista: Utilizar a facilidade dos procedimentos com um número não fixo de argumentos

Exercício 4.21
Escrever o procedimento *repetir-fun* que toma como argumentos uma função numérica, *f*, e um inteiro positivo, *n*, e devolve a função `f ( f ( ... ( f ( x ) )... ) )`, ou seja, a função *f* aplicada sobre si mesma *n* vezes.

```
↦((repetir-fun add1 3) 5)       ; equivalente a (add1 (add1 (add1 5)))
8
```

Exercício 4.22
Desenvolver o procedimento *raiz*, tomando por base outros procedimentos já desenvolvidos, nomeadamente, *repetir-fun* (exercício anterior) e *newton* (secção 3 do presente capítulo), que apresenta como parâmetros *n* e *num* e devolve a raiz de ordem *n* de *num*.

```
↦(raiz 2 49)
7
↦(raiz 5 243)
3
```

Exemplo 4.8

Escrever o procedimento *faz-ordenador* que aceita como argumento um operador de comparação de dois elementos e devolve um procedimento com um único parâmetro, que representa uma lista. O procedimento devolvido, quando chamado, ordena os elementos da lista que aceita como argumento, de acordo com o operador de comparação. As chamadas que se seguem pretendem ajudar a esclarecer o que foi dito sobre *faz-ordenador*.

```
↦(define ordenador-crescente (faz-ordenador <))
ordenador-crescente
↦(ordenador-crescente '(23 5 37 1 -8))
(-8 1 5 23 37)
↦(define ordenador-decrescente (faz-ordenador >))
ordenador-decrescente
↦(ordenador-decrescente '(23 5 37 1 -8))
(37 23 5 1 -8)
```

A solução que se apresenta merece algumas notas:

⇒ Como facilmente se verifica, *faz-ordenador* devolve um procedimento com um parâmetro que é uma lista e cujo corpo se limita a uma chamada do procedimento *ordena*;

⇒ No passo recursivo do procedimento *ordena*, é chamado o procedimento *insere-por-comparacao*, o qual insere um elemento numa lista previamente ordenada, continuando ordenada após a inserção do elemento. Isto poderá parecer estranho, mas não é mais do que a recursividade a funcionar. De facto, o terceiro argumento da chamada deste procedimento, é `(ordena comparacao (cdr lista))` que deverá devolver a lista *`(cdr lista)`* devidamente ordenada...

⇒ Caberá ao programador decidir entre construir ordenadores com o procedimento *faz-ordenador* ou então utilizar directamente o procedimento *ordena*, indicando qual o operador de comparação que pretende.

Solução

```
(define faz-ordenador
  (lambda (comparacao)

    (lambda (lista)
      (ordena comparacao lista))))


(define ordena
  (lambda (comparacao lista)
    (if (null? lista)
        '()
        (insere-por-comparacao (car lista)
                               comparacao
                               (ordena comparacao (cdr lista))))))

(define insere-por-comparacao   ; insere um elemento numa lista ordenada,
  (lambda (elem compara lis-ordenada)   ; continuando ordenada, após a inserção
    (cond ((null? lis-ordenada)(list elem))
          ((compara elem (car lis-ordenada))
           (cons elem lis-ordenada))
          (else
            (cons (car lis-ordenada)
                  (insere-por-comparacao
                    elem
                    compara
```

```
                    (cdr lis-ordenada)))))))
```

Exercício 4.23
Para se obter uma solução iterativa do procedimento considerado no exemplo anterior, agora designada por *faz-ordenador-iter*, completar o que se segue, escrevendo o procedimento *ordena-iter*.

```
(define faz-ordenador-iter
  (lambda (comparacao)

    (lambda (lista)
      (ordena-iter comparacao lista '()))))

(define ordena-iter
  (lambda (comparacao lista lista-ordenada)

... para completar ...
```

Pista: *lista-ordenada* encontra-se inicialmente vazia, como se pode verificar pela primeira chamada de *ordena-iter*. À medida que surgem elementos de *lista* para ordenar, cada um deles é inserido em *lista-ordenada*, na posição correcta, de acordo com a regra de ordenação. Assim, garante-se que *lista-ordenada* encontra-se sempre ordenada. A inserção de mais um elemento numa lista já ordenada pode ser conseguida pela utilização do procedimento *insere-por-comparacao*, do exemplo anterior.
Contrariamente ao que é normal, a solução iterativa surge mais fácil de entender do que a solução recursiva.

Exercício 4.24
Escrever em *Scheme* o procedimento *faz-funcao-acesso-a-lista* que aceita como argumento um inteiro positivo e devolve um procedimento com um único parâmetro, que representa uma lista. O procedimento devolvido, quando chamado, acede e devolve um elemento da lista, conforme se pode verificar nas chamadas que se seguem.

```
↦(define terceiro (faz-funcao-acesso-a-lista 2))
terceiro      ; o terceiro elemento de uma lista está na posição 2, considerando
              ; que na posição zero está o primeiro elemento. Por isso, na criação
              ; de terceiro, foi utilizado 2 como argumento de faz-funcao-acesso-a-lista

↦(terceiro '(p0 p1 p2 p3 p4 p5 p6 p7))
p2
↦(define setimo (faz-funcao-acesso-a-lista 6))
setimo
↦(setimo '(p0 p1 p2 p3 p4 p5 p6 p7))
p6
↦(define decimo (faz-funcao-acesso-a-lista 9))
decimo
↦(decimo '(p0 p1 p2 p3 p4 p5 p6 p7))
erro: lista pequena
```

Projecto 4.1 - CODEC

Este projecto relaciona-se com o desenvolvimento de um conjunto de procedimentos para codificar e descodificar mensagens. Para este efeito considera-se que uma mensagem é uma lista de símblos formados por um letra, como, por exemplo, '(a d f e g h i).
A codificação baseia-se na substituição dos símbolos originais por outros, de acordo com um código preparado manualmente:

```
↦(define codigo-1 '((a 1) (b 2) (c 3)))
codigo-1
↦(define codigo-2 '((a b) (b t) (c 4) (d 1)))
codigo-2
```

Segundo *codigo-1*, o alfabeto a utilizar nas mensagens é composto pelas letras *a*, *b*, *c*, às quais corresponde a codificação *1*, *2*, e *3*, respectivamente. Qualquer letra fora do alfabeto é substituída, na mensagem codificada, por *'erro*. Para *codigo-2*, o alfabeto utilizado abarca as 4 primeiras letras do abecedário, às quais corresponde a codificação *b*, *t*, *4*, e *1*, respectivamente.
O procedimento *faz-codificador* aceita apenas um argumento que deverá ser um código, previamente definido, e devolve um procedimento com um único parâmetro. A este parâmetro, em cada chamada, deverá ser associada a mensagem a codificar.

```
↦(define codificador-1 (faz-codificador codigo-1))
codificador-1
↦(codificador-1 '(a b c b a))
(1 2 3 2 1)
↦(codificador-1 '(a b c b x y a))
(1 2 3 2 erro erro 1)
↦(define codificador-2 (faz-codificador codigo-2))
codificador-2
↦(codificador-2 '(a b c b a))
(b t 4 t b)
↦(codificador-2 '(a b c b x y a))
(b t 4 t erro erro b)
↦(codificador-1 (codificador-2 '(a b c b a)))
(2 erro erro erro 2)
```

Analisar agora as chamadas que se seguem, relacionadas com a descodificação de mensagens.

```
↦(define descodificador-1 (faz-descodificador codigo-1))
descodificador-1
↦(descodificador-1 '(1 2 3 2 1))
(a b c b a)
↦(descodificador-1 '(1 2 3 2 erro erro 1))
(a b c b erro erro a)
```

1- Escrever em *Scheme* os procedimento *faz-codificador* e *faz-descodificador*.
2- Para uma maior segurança, os códigos passarão a ser criados automaticamente. Para isso, escrever o procedimento *cria-codigo*, com o parâmetro *alfabeto*, lista com os caracteres de um alfabeto, e que devolve um código gerado aleatoriamente (o formato é semelhante ao dos códigos utilizados). No entanto, o procedimento deverá garantir que nos códigos criados não haverá caracteres do alfabeto com a mesma codificação. Vai-se considerar que a codificação de qualquer caracter será sempre um caracter do próprio alfabeto.

```
↦(define c1 (cria-codigo '(a b c 1 2 3)))
c1                ; verificar que, de facto, o código de qualquer caracter de c1 é um
↦c1                                  ; caracter do alfabeto fornecido a cria-codigo
((a 1) (b a) (c 3) (1 2) (2 c) (3 b))
↦(define codificador-1 (faz-codificador c1))
```

```
codificador-1
↦(define descodificador-1 (faz-descodificador c1))
descodificador-1
↦(codificador-1 '(a b c d e 4 3 2 1))
(1 a 3 erro erro erro b c 2)
↦(descodificador-1 '(1 a 3 erro erro erro b c 2))
(a b c erro erro erro 3 2 1)
↦(descodificador-1 (codificador-1 '(a b f 2 2 5 c 1 3)))
(a b erro 2 2 erro c 1 3)
```