Módulo 4.3 - **Resumo dos módulos 4.1 e 4.2, exemplos e exercícios**

Resumo dos assuntos abordados nos módulos
Módulo 4.1 - **Os procedimentos não param de surpreender**
Módulo 4.2 - **Um procedimento pode devolver outro procedimento**

Após o resumo são apresentados alguns exemplos e exercícios para treino da matéria relacionada com os módulos acima referidos.

A utilização do Scheme é fundamental neste módulo.

Resumo dos assuntos abordados nos módulos 4.1 e 4.2

Neste conjunto de dois módulos, o objectivo principal é mostrar que os procedimentos em Scheme também são *objectos de 1ª classe*, objectos que podem ser manipulados, como acontece com os números, booleanos, símbolos e pares. As características dos *objectos de 1ª classe* aplicam-se aos procedimentos, pois os procedimentos ligam-se a identificadores, são argumentos de outros procedimentos, são elementos de estruturas de dados e são valores de retorno de outros procedimentos.
Num pequeno desvio inicial, mostra-se como se definem procedimentos que podem ser chamados com um número não fixo de argumentos.

Módulo 4.1
As propriedades que caracterizam os *objectos de 1ª classe* também se aplicam aos procedimentos e a comprovação deste este facto inicia-se neste módulo.
Este módulo começa com um pequeno desvio ao tema e apresenta mais uma novidade sobre os procedimentos - os procedimentos podem admitir um número não fixo de argumentos. Ou seja, um procedimento pode ser chamado com qualquer número de argumentos, podendo inclusivamente ser chamado com zero argumentos. Depois deste desvio, entra-se numa primeira abordagem aos procedimentos como objectos de 1ª classe, mostrando duas das suas características: são argumentos de outros procedimentos e são elementos de estruturas de dados. Neste percurso, surgem naturalmente os procedimentos que aceitam outros procedimentos como argumentos, designados por *procedimentos de ordem mais elevada* e deles são apresentados alguns exemplos tanto como procedimentos primitivos (*map*, *for-each* e *apply*), como procedimentos definidos no decorrer dos módulos (*max-fun* e *somatorio*).

Módulo 4.2
Os procedimentos foram frequentemente associados a identificadores (identificadores que passaram a ser o nome desses procedimentos), foram utilizados e desenvolvidos procedimentos que aceitavam outros procedimentos como argumentos (procedimentos de ordem mais elevada) e alguns procedimentos fizeram parte de estruturas de dados. Todas estas características foram comprovadas no módulo anterior.
Neste módulo, surge mais uma características dos procedimentos - podem ser valor de retorno de outros procedimentos. Com esta característica dos procedimentos completa-se o conjunto de características dos *objectos de 1ª classe*.

Exercícios e exemplos

São apresentados alguns exercícios e exemplos para consolidação da matéria tratada nos módulos 4.1 e 4.2.

Exemplo 1
O procedimento primitivo *max* aceita um número não fixo de argumentos e devolve o valor do maior argumento. Escreva em Scheme uma versão pessoal de *max*, designada por *max-pessoal*.

```
(define max-pessoal-v1
  (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)))))
```

Esta primeira versão de *max-pessoal* baseia-se no procedimento interno *max-aux* que gera processos recursivos, com *O(n)* em tempo e espaço, em que *n* está relacionado com o número de valores a processar. A ideia utilizada é, à partida, muito simples, mas não é fácil de acompanhar: toma-se o primeiro valor a processar que é comparado com o maior valor encontrado entre os restantes...
A versão seguinte baseia-se numa ideia simultaneamente simples e muito fácil de acompanhar: o primeiro valor é considerado o maior valor até essa altura e torna-se o *máximo-temporário*; o *máximo-temporário* é comparado com o valor seguinte, e o maior entre eles passa a ser o *máximo-temporário*; a comparação repete-se até não haver mais valores, estando a resposta no *máximo-temporário*.

```
(define max-pessoal-v2
  (lambda args
    (letrec ((max-aux-iter
               (lambda (lis max-temp)
                 (cond
                   ((null? lis)
                    max-temp)
                   ((> (car lis) max-temp)
                    (max-aux-iter (cdr lis) (car lis)))
                   (else
                    (max-aux-iter (cdr lis) max-temp))))))

      (if (null? args)
          (display "erro. Numero incorrecto de argumentos")
          (max-aux-iter (cdr args) (car args))))))
```

Espaço para testar as duas versões do procedimento *max-pessoal*.
O ecrã abre com as duas versões referidas.

Indique e justifique a ordem de complexidade dos processos gerados pela versão max-pessoal-v2.

Seguir, passo a passo, o funcionamento das duas versões de max-pessoal para a chamada `(max-... 2 7 1 3)`.

Exercício 1
Escreva em Scheme o procedimento *pi-aprox*, baseado em *somatorio*, partindo da fórmula:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

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

Espaço para desenvolver e testar o procedimento *pi-aprox*. O ecrã abre com o procedimento *somatorio* e com identificador *pi* assim definido: `(define pi 3.141592653589793)`

Exercício 2

O valor de *e* pode ser aproximado através da fórmula $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$

Escreva em Scheme o procedimento *e-aprox*, com o parâmetro *n*, e devolve o valor de *e* aproximado aos primeiros *n* termos da fórmula.

Espaço para desenvolver e testar o procedimento *e-aprox*. O ecrã abre com o procedimento *somatorio* e com o identificador *e* assim definido: `(define e 2.71828182846)`

Exercício 3

O *golden ratio* pode ser definido pela fórmula $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

e calculado através de $\Phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}}$

Escreva em Scheme o procedimento *golden-aprox*, com o parâmetro *n*, que devolve o valor do *golden ratio* aproximado aos primeiros *n* termos da fórmula.

Espaço para desenvolver e testar o procedimento *golden-aprox*. O ecrã abre com o procedimento *somatorio*.

Exercício 4

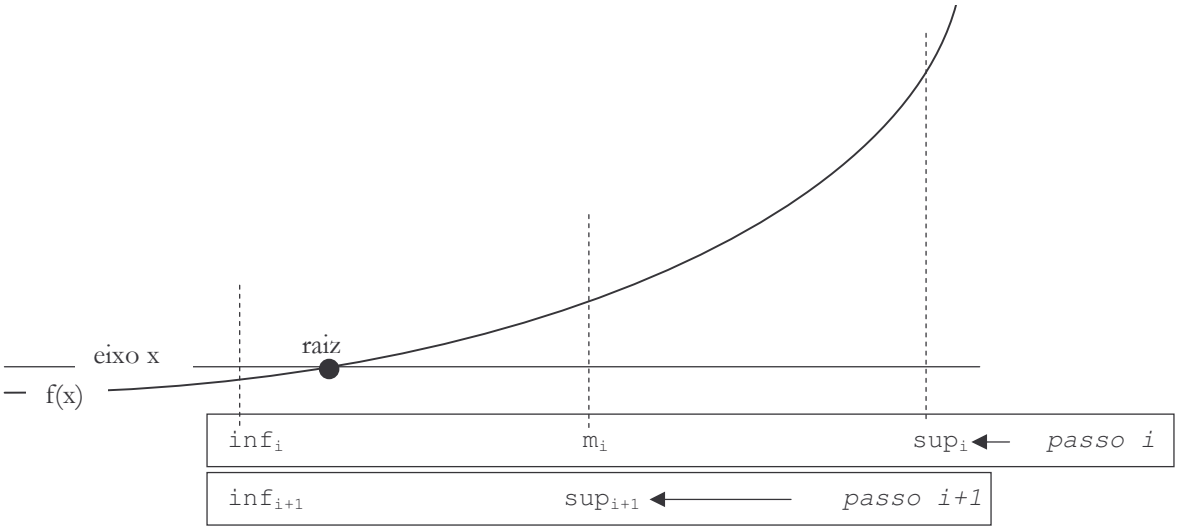
A fórmula que se apresenta é uma aproximação do integral de uma função *f(x)*, entre *a* e *b*. Verifique, graficamente, que a aproximação é tanto melhor quanto menor for *dx*.
Neste contexto, escreva em Scheme o procedimento *integral*, baseado em *somatorio*.

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

Espaço para desenvolver e testar o procedimento *integral*. O ecrã abre com o procedimento *somatorio*.

Exercício 5

O *método da bissecção*, para encontrar as raízes de funções, é esquematizado na figura.



No *passo i* do processo de procura, a raiz encontra-se no domínio que tem por limite inferior `x = infi` e por limite superior `x = supi`, pois sabe-se que `f(infi) < 0 < f(supi)`. É, então, determinado o ponto médio do referido limite, m_i, e feitas as seguintes verificações sequencialmente:

- Se `f(mi)` é suficientemente próximo de zero, a raiz é tomada como sendo m_i;
- Se `f(mi) > 0`, a procura continua, no *passo i+1*, com `infi+1 = infi` e `supi+1 = mi`;
- Se `f(mi) < 0`, a procura continua, no *passo i+1*, com `infi+1 = mi` e `supi+1 = supi`.

Escreva em Scheme um *procedimento de ordem mais elevada*, com a designação *bisseccao*, que implementa a técnica acabada de apresentar, com os parâmetros *fun*, *inf* e *sup*, em que *fun* representa uma função `y=f(x)`, e *inf* e *sup* definem os limites do domínio em que se procura a raiz, havendo a garantia que `f(inf) < 0 < f(sup)`.

Espaço para desenvolver e testar o procedimento *bisseccao*. O ecrã abre vazio.

Exemplo 2

Escreva em Scheme o procedimento *compoe* que tem os procedimentos *p1* e *p2* como parâmetros e devolve um procedimento de um só parâmetro, *x*. O procedimento resultante aplica *p1* a `p2(x)`, ou seja, `p1(p2(x))`.

```
(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 9)
100
```

Espaço para testar o procedimento *compoe*. O ecrã abre com este procedimento.

Exercício 6
Escreva em Scheme o procedimento *compoe3* que tem os procedimentos *p1*, *p2* e *p3* como parâmetros e devolve um procedimento de um único parâmetro, *x*. O procedimento resultante faz `p1(p2(p3(x)))`.

Espaço para desenvolver e testar o procedimento *compoe3*. O ecrã abre com o procedimento *compoe*.

Exercício 7
Definir o procedimento *compoe-muitas*, que responde da seguinte maneira:

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

Espaço para desenvolver e testar o procedimento *compõe-muitas*. O ecrã abre com o procedimento *compoe*.

Pista: Utilizar procedimentos com um número não fixo de argumentos.

Exercício 8
Escreva em Scheme 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)          (é equivalente a
8                                            (add1 (add1 (add1 5))) )
```

Espaço para desenvolver e testar o procedimento *repetir-fun*. O ecrã abre com o procedimento *compoe*.

Exemplo 3
Analise as chamadas de *newton-raphson* que se seguem.
Na primeira, determina-se o valor de *x* para o qual $x^2 - 9 = 0$, equivalente a $x = \sqrt{9}$. O zero da função $f(x) = x^2 - 9$ coincide com a raiz quadrada de 9.

```
> (newton-raphson (lambda (x) (- (* x x) 9)) 1)
3.0000000174227237
```

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

```
> (newton-raphson (lambda (x) (- (* x x x) 9)) 1)
2.080083834331853
```

Finalmente, na terceira, o zero da função $f(x) = x^3 - 8$ coincide com a raiz cúbica de 8.

```
> (newton-raphson (lambda (x)(- (* x x x) 8)) 1)
2.0000000033068637
```

Destes exemplos deduz-se uma ideia para a escrita de procedimentos que calculam a raiz quadrada e a raiz 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)
6.928203438125418
> (sqrt 48)             (o mesmo cálculo recorrendo directamente a
6.928203230275509                        um procedimento do Scheme)
```

Espaço para testar os procedimentos *raiz-quadrada* e *raiz-cubica*. O ecrã abre com estes procedimentos e ainda com o procedimento *newton-raphson*.

Exercício 9 - para pensar um bocado...
Desenvolver o procedimento *raiz*, tomando por base outros procedimentos já desenvolvidos, nomeadamente, *repetir-fun* (exercício anterior) e *newton-raphson*. O procedimento *raiz* apresenta como parâmetros *n* e *num* e devolve a raiz de ordem *n* de *num*.

```
> (raiz 2 49)                    (é equivalente a x = √49 )
7
> (raiz 5 243)                   (é equivalente a x = ⁵√243 )
3
```

Espaço para desenvolver e testar o procedimento *raiz*. O ecrã abre com os procedimentos *repetir-fun* e *newton-raphson*.

Exemplo 4
Escreva em Scheme 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 ajudam a esclarecer o que foi dito sobre *faz-ordenador*.

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

A solução que se apresenta merece algumas notas:

- *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...

```
(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))))))

; insere um elemento numa lista ordenada,
; continuando ordenada, após a inserção
(define insere-por-comparacao
  (lambda (elem compara lis-ordenada)
    (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)))))))
```

Espaço para testar o procedimento *faz-ordenador*.
O ecrã abre com este procedimento e com os procedimentos auxiliares *ordena* e *insere-por-comparacao*.

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.
Comente esta opinião.

Exercício 10

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 ...
```

Espaço para testar o procedimento *faz-ordenador-iter*.
O ecrã abre com o código indicado para completar.

Pista: O parâmetro *lista-ordenada* é uma lista inicialmente vazia, como se verifica na 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 apresenta-se mais fácil de entender do que a solução recursiva, do exemplo anterior.
Concorda?

Exercício 11

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))
```

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

Espaço para desenvolver e testar o procedimento *faz-funcao-acesso-a-lista*.
O ecrã abre vazio.

Exercício 12

Projecto - CODEC

Pretende-se desenvolver um conjunto de procedimentos para codificar e descodificar mensagens. Para este efeito, considera-se que uma mensagem é uma lista de símbolos 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)))
> (define codigo-2 '((a b) (b t) (c 4) (d 1)))
```

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, pelo símbolo *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 '(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 '(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)
```

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

```
> (define descodificador-1 (faz-descodificador codigo-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- Escreva em Scheme os procedimento *faz-codificador* e *faz-descodificador*.
2- Para uma maior segurança, os códigos passarão a ser criados automaticamente. Para isso, escreva o procedimento *cria-codigo*, com o parâmetro *alfabeto*, lista com os caracteres de um alfabeto, 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. Considere que a codificação de qualquer carácter será sempre um carácter do próprio alfabeto.

```
> (define c1 (cria-codigo '(a b c 1 2 3)))
> c1
((a 1)(b a) (c 3) (1 2) (2 c) (3 b))
> (define codificador-1 (faz-codificador c1))
> (define descodificador-1 (faz-descodificador c1))
> (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)
```

Espaço para desenvolver e testar o projecto CODEC.
O ecrã abre vazio.