Módulo 2.1 - **Resolver problemas, passo a passo, na direcção de um caso conhecido!**

Na definição de um procedimento podem surgir chamadas a outros procedimentos, situação perfeitamente normal e fácil de entender. No entanto, já não parece tão natural que um procedimento se chame a si próprio! Através de um raciocínio mais ligeiro, até poderia parecer que este procedimento nunca mais terminaria... Falamos de *Recursividade*, a qual está na base da definição dos *procedimentos recursivos*, isto é, procedimentos que se chamam a si próprios e que nos irão surpreender pelas soluções simples e elegantes que proporcionam.

O objectivo deste módulo é mostrar como se definem procedimentos recursivos e mostrar como este tipo de procedimentos resolvem problemas, caminhando passo a passo, na direcção de um caso conhecido!
É recorrendo à *Recursividade* que o Scheme implementa tarefas repetitivas, pois não possui as estruturas de repetição, como acontece noutras linguagens de programação (tais como *for* ou *while*).
Atenção, pois estamos perante um tema muito importante e que é necessário dominar na perfeição.

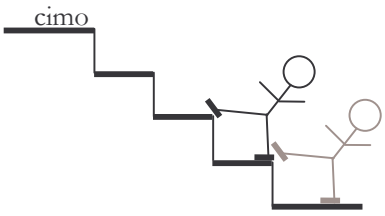
Palavras-Chave
recursividade, tarefa recursiva, caso base ou condição de terminação, operação de redução, caso geral ou passo recursivo, exprimir a recursividade em Scheme.

Passo a passo, na direcção de um caso conhecido

O desafio que se coloca, para já, é a análise de situações do nosso quotidiano, resolvidas tão naturalmente, que quase não damos por elas.

Se pretender subir umas escadas até ao cimo, procederá quase sem pensar, da seguinte maneira:

Tarefa *subir as escadas*
- Se atingiu o cimo das escadas, terminou a tarefa *subir as escadas*
- Pelo contrário, se ainda não atingiu o cimo
  - avança um degrau na direcção do cimo das escadas e
  - retoma a tarefa *subir as escadas*



Nota, certamente, algo de estranho nesta forma de actuar, pois no interior da tarefa *subir as escadas*, esta mesma tarefa chama-se a si própria.
Acabou por identificar um exemplo de uma tarefa *recursiva* ou, de outra forma, está perante uma forma de actuar que faz uso de *Recursividade*.

Como pode garantir que esta tarefa recursiva tem fim?
Mesmo que tenha que dispensar algum tempo, procure por si dois factores que, em conjunto, dão esta garantia.

A garantia de que esta tarefa recursiva tem fim resulta da conjugação de dois factores:

- Começa sempre por testar a condição de terminação (*já se atingiu o cimo das escadas?*) e, em caso afirmativo, a tarefa termina.
- Antes de cada nova chamada da tarefa recursiva, a dimensão do problema é reduzida (*avançar um degrau na direcção do cimo das escadas*) e assim fica mais próximo do fim.

Numa tarefa recursiva há sempre três aspectos que a caracterizam e que deve identificar de uma forma muito clara

- *Caso base* ou *Condição de terminação - já atingiu o cimo das escadas?*
- *Operação de redução* da dimensão do problema - *avança um degrau na direcção do cimo das escadas*
- *Caso geral* ou *Passo recursivo - retoma a tarefa subir as escadas, depois de ter subido um degrau*

Tome atenção ao exemplo que se segue, pois vai ter que identificar os três aspectos que caracterizam a respectiva tarefa recursiva. Suponha que tem um saco com moedas e que pretende determinar a quantia em dinheiro nele contida.

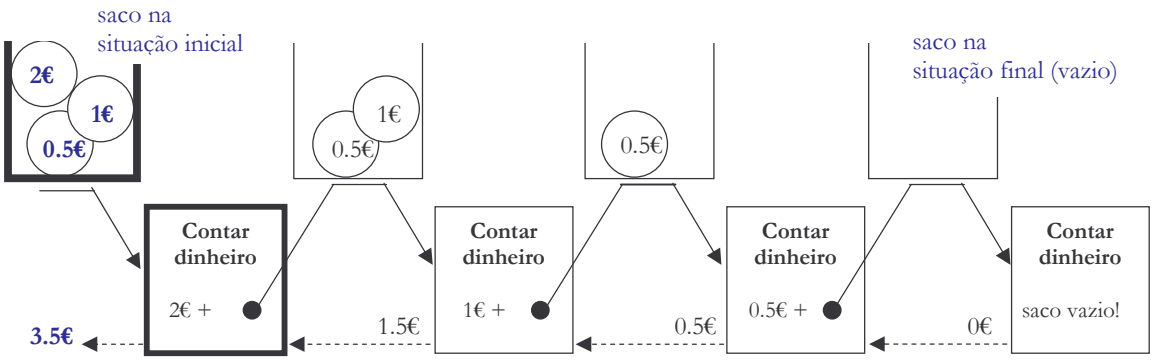
Tarefa *determinar a quantia contida no saco*

- Se o saco estiver vazio, sem moedas, isso corresponde à quantia 0€
- Pelo contrário, se o saco contiver moedas
  - retira uma moeda do saco
  - adiciona o valor da moeda retirada com a quantia em dinheiro que ainda continua no saco, o que requer retomar a tarefa *determinar a quantia contida no saco*

Também aqui, ao retomar a tarefa determinar a quantia contida no saco, a dimensão da tarefa surge mais reduzida. Justifique.

Neste exemplo,

- *Caso base* ou *Condição de terminação - saco vazio?*, a que corresponde a quantia de *0€*
- *Operação de redução* da dimensão do problema - *retira uma moeda* do saco
- *Caso geral* ou *Passo recursivo - adiciona o valor da moeda retirada com a quantia que ainda resta no saco*



Observe na figura que a contagem do dinheiro que se encontra no saco começa com o lançamento do processo *Contar dinheiro*, representado pelo rectângulo mais à esquerda.

Este processo de contagem identifica o valor da moeda retirada, mas necessita também de conhecer o valor corresponde às moedas que ainda se encontram no saco. Por este facto, e reconhecendo que esta tarefa é idêntica à que procura resolver, lança um novo processo de contagem, agora com o saco com menos uma moeda. Isto é representado pelo segundo rectângulo a contar da esquerda.
Um aspecto muito importante é o seguinte, o processo correspondente ao rectângulo mais à esquerda vai ficar suspenso até que lhe devolvam a informação pedida, ou seja, o valor corresponde às moedas ainda no saco.
O segundo processo, pelas razões expostas para o primeiro, vai também lançar um novo processo e fica suspenso também à espera que lhe devolvam a informação pedida.
E esta sequência de lançamento de novos processos continua até ao processo que detecta que o saco está vazio. Este é o único que conhece imediatamente a quantia que se encontra no saco, 0€, e por isso devolve a informação respectiva ao processo que o lançou. Este processo, representado pelo rectângulo mais à direita, não fica suspenso, pois resolveu completamente a tarefa que lhe foi solicitada. Por isso morre definitivamente e liberta os recursos computacionais de memória que utilizava.
Entretanto, o processo suspenso que recebe a informação de 0€ pode agora completar a sua tarefa devolvendo a quantia de 0€ + 0.5€ = 0.5€. Assiste-se assim ao finalizar dos processos suspensos. Quando o processo mais à esquerda recebe a informação correspondente à quantia de 1.5€, já todos os outros processos morreram. E também ele morrerá, logo após ter devolvido a quantia de 1.5€ + 2€ = 3.5€, o valor da totalidade das moedas contidas no saco.

Como pode observar, um processo lançado vai ficar suspenso à espera de uma resposta que lhe será dada por um outro processo por ele lançado, e isto repete-se até que se encontre um caso conhecido.
Imagine que o saco era enorme e que tinha mil moedas.
Que recursos computacionais poderiam esgotar? Justifique.

Exercício 1
Imagine duas situações da vida real que possam ser resolvidas recorrendo à recursividade, e identifique para cada uma delas o *Caso base* ou *Condição de terminação*, a *Operação de redução* e o *Caso geral* ou *Passo recursivo*.

Exprimir a recursividade em Scheme

O *factorial* de um inteiro positivo *n* é o exemplo clássico utilizado para ilustrar a recursividade. É representado por `n!` = `n * (n-1) * (n-2) * ... * 3 * 2 * 1`.

Vejamos o valor de *factorial* para várias situações de *n*.

```
0! = 1 (por convenção)
1! = 1                                  = 1 * 0!
2! = 2 * 1                              = 2 * 1!
3! = 3 * 2 * 1                          = 3 * 2!
4! = 4 * 3 * 2 * 1                      = 4 * 3!
...
n! = n * (n-1) * (n-2) * ... * 1   = n * (n - 1)!
```

Já pode daqui retirar as características desta tarefa recursiva:

Caso base ou *Condição de terminação* n igual a 0 `0! = 1` (definido por convenção)

Operação de redução	subtrair 1 a n	n - 1
Caso geral ou *Passo recursivo*	n diferente de 0	`n! = n * (n-1)!`

Notar que calcular (n-1)!, relativamente ao caso que pretende resolve, que é n!, representa uma redução da dimensão do problema, pois (n-1)! está mais perto do *Caso base* do que n!.

A partir da definição de *factorial*, já pode escrever o respectivo procedimento em Scheme.

```
; Procedimento recursivo com um parâmetro, num.
; Devolve o factorial de num.
;
(define factorial
  (lambda (num)
    (if (zero? num)
        1                                   ; o caso base
        (* num (factorial (sub1 num))))))   ; o caso geral

> (factorial 0)
1
> (factorial 4)
24
```

Espaço para experimentar o procedimento recursivo *factorial*.
O ecrã abre com este procedimento e com o diálogo indicado.

Por que razão, na sua opinião, na primeira linha dos comentários associados ao procedimento factorial, é dito que o procedimento é recursivo?

Para melhor entender como funciona este procedimento recursivo, siga, passo a passo, o desenrolar da chamada `(factorial 4)` a que corresponde a resposta 24.

Seguindo a expressão *if* do procedimento *factorial* e uma vez que num = 4, portanto, diferente de zero

```
(factorial 4) = (* 4 (factorial (sub1 4)))
              = (* 4 (factorial 3))
```

Um dos operandos da expressão obtida é `(factorial 3)`, cujo cálculo implica nova chamada ao procedimento *factorial*, mas agora com num = 3, também diferente de zero. Seguindo novamente a expressão *if*:

```
(factorial 4) = (* 4 (factorial 3))
              = (* 4 (* 3 (factorial (sub1 3))))
              = (* 4 (* 3 (factorial 2)))
```

O comprimento da expressão vai aumentando, mas, em contrapartida, as chamadas ao procedimento *factorial* vão-se aproximando do *Caso base*, o único de que se conhece a solução, por convenção. Continuando a seguir a chamada a *factorial* com num = 2, depois com num = 1 e, finalmente, com num = 0:

```
(factorial 4) = (* 4 (* 3 (* 2 (* 1 (factorial 0)))))
```

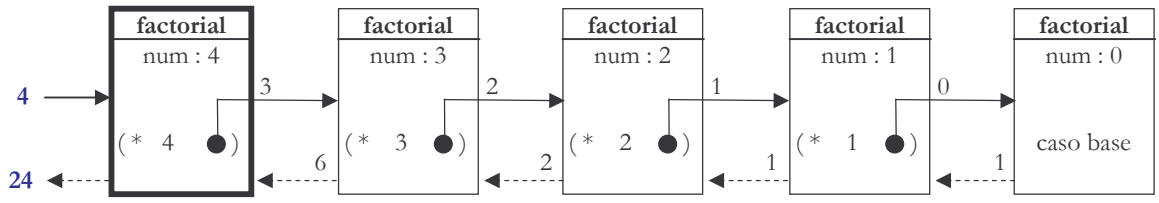
Observe que as operações de multiplicação vão ficando suspensas.
Poderá daqui concluir que um procedimento recursivo impõe sempre uma reserva de recursos computacionais de memória, que poderão ser enormes para dimensões grandes do problema que resolve (número de moedas do saco ou valor n)?

Neste ponto, atingiu-se o *Caso base*, 0! = 1, e a expressão vai começar a reduzir, pois os processos suspensos vão começar a receber a informação que pediram.

```
(factorial 4) = (* 4 (* 3 (* 2 (* 1 (factorial 0)))))
              = (* 4 (* 3 (* 2 (* 1 1))))
              = (* 4 (* 3 (* 2 1)))
              = (* 4 (* 3 2))
              = (* 4 6)
              = 24
```

Identifique a multiplicação que esteve mais tempo suspensa?
Caso não consiga, preste atenção ao texto e à figura que se seguem...

O funcionamento do procedimento *factorial*, para a chamada `(factorial 4)`, pode também ser observado na figura, que deverá começar a ser analisada da esquerda para a direita.
Enquanto não se atinge o *Caso base* (representado pelo rectângulo mais à direita), vão sendo gerados



processos todos eles para realizarem tarefas idênticas, mas de dimensão sucessivamente mais pequena. A chamada `(factorial 4)` gera o processo representado pelo rectângulo mais à esquerda, em que num = 4. Este processo, ao calcular `(* num (factorial 3))`, gera um novo processo correspondente à chamada `(factorial 3)` e suspende a sua actividade até obter uma resposta. A geração sequencial de novos processos continuará até se atingir o *Caso base*, com num = *0*. Este processo pode imediatamente responder, devolvendo o valor 1. Observa-se agora que os processos suspensos vão poder terminar a sua tarefa, o que ocorrerá pela ordem inversa à criação desses mesmos processos. Assim, o primeiro processo criado para responder à chamada `(factorial 4)` será o último a terminar, situação que ocorre quando devolver o valor 24, depois de ter recebido o valor 6, do processo anterior.

É agora possível delinear a estratégia para escrever *procedimentos recursivos*:

- Identifique o *Caso base* ou *Condição de terminação*, caso para o qual se conhece a solução do problema. No exemplo *factorial*, a condição de terminação é num = 0, para a qual 0! = 1
- Identifique a *Operação de redução* a aplicar sucessivamente, até se atingir o *Caso base*. No mesmo exemplo, a operação de redução é *sub1* aplicada a *num*
- Escreva o procedimento recursivo começando pelo *Caso base*, passando seguidamente ao *Caso geral*. O *Caso geral* envolve uma chamada recursiva ao procedimento que se pretende definir, chamada essa que deverá aproximar-se do *Caso base*. Ainda no exemplo *factorial*, na situação correspondente a *num*, a chamada recursiva passaria a ser sobre `(sub1 num)`

Exemplo 1
Escreva em Scheme o procedimento *soma-digito*s que aceita um argumento inteiro e positivo, *numer*o, e devolve a soma dos dígitos decimais do argumento.

Vamos começar com o desenvolvimento manual de algumas situações de funcionamento do procedimento pretendido.

```
> (soma-digitos 823)
13
> (soma-digitos 000000)
0
```

```
> (soma-digitos -823)
Numero negativo!!!
```

Para definir este procedimento, a ideia a explorar resume-se a somar os dígitos do argumento, um a um, começando pelo menos significativo (o da direita) até atingir o mais significativo (o da esquerda). Para aplicar a recursividade, toma-se o dígito menos significativo que será somado à soma dos dígitos restantes, começando assim uma operação de redução do problema. Por exemplo,

```
(soma-digitos 823) = (+ 3 (soma-digitos 82))
```

Neste exemplo, a dimensão inicial do problema é 3, pois o número 823 é composto por 3 dígitos. Depois da redução, passou a um problema de dimensão 2, correspondente ao número 82. A identificação do *caso base* é fácil. Perante um número inferior a 10, portanto, com um só dígito, a soma dos dígitos desse número é conhecida, pois é o próprio número.

```
(soma-digitos 823) = (+ 3 (soma-digitos 82))
                   = (+ 3 (+ 2 (soma-digitos 8)))     <--- caso base

                   = (+ 3 (+ 2 8))
                   = (+ 3 10)
                   = 13
```

Não esqueça, neste exemplo, que é necessário verificar se o número é negativo e, se assim for, é apenas visualizada a mensagem correspondente.
Mas suponha, para já, que o número é positivo.

- O *caso base* - Número menor que 10, pois a resposta torna-se imediatamente conhecida. A soma dos dígitos é o próprio número
- A *operação de redução* - Quociente da divisão inteira (*quotient* em Scheme) do número por 10 , que equivale a retirar o dígito menos significativo ao número, reduzindo assim a dimensão do problema.
- O *caso geral* - Somar o dígito menos significativo à soma dos dígitos do número entretanto reduzido. O dígito menos significativo poderá ser obtido através do resto da divisão inteira (*remainder* em Scheme) do número por 10.

Na solução que se segue foram identificadas duas tarefas auxiliares em que se decompõe a tarefa *soma-digitos*, designadas por *digito-menos-significativo* e *retira-digito-menos-significativo*.

```
(define digito-menos-significativo
  (lambda (n)
    (remainder n 10)))              ; o resto é o dígito menos significativo

> (digito-menos-significativo 823)
3


(define retira-digito-menos-significativo
  (lambda (n)                       ; deste quociente resulta o número
     (quotient n 10)))              ; sem o dígito menos significativo

> (retira-digito-menos-significativo 823)
82
```

Espaço para experimentar os procedimentos auxiliares *digito-menos-significativo* e *retira-digito-menos-significativo*.
O ecrã abre com estes procedimentos e com o diálogo indicado.

Finalmente, a definição e teste do procedimento *soma-digitos*.

```
(define soma-digitos
  (lambda (num)
```

```
    (cond ((negative? num)
           (display "Numero negativo!!!"))
          ;
          ((< num 10) num)            ; o caso base
          ;
          (else (+ (digito-menos-significativo num)     ; o caso geral
                   (soma-digitos (retira-digito-menos-significativo num)))))))
```

```
> (soma-digitos 823)
13
> (soma-digitos -823)
Numero negativo!!!
> (soma-digitos 1234567890)
45
>
```

Espaço para experimentar o procedimento *soma-digitos* e os procedimentos auxiliares *digito-menos-significativo* e *retira-digito-menos-significativo*. O ecrã abre com estes procedimentos e com o diálogo indicado.

Exercício 2

Verifique que, em *soma-digitos*, o teste *negative?* é repetido sucessivas vezes, quando, na realidade, apenas seria necessário da primeira vez.

O teste referido quantas vezes tem lugar inutilmente?

Para resolver esta situação, bem como outras do mesmo tipo, basta que imagine um procedimento auxiliar que trate apenas o caso ideal, em que o argumento nunca é negativo. Seja esse o procedimento o *soma-digitos-de-positivo*:

```
(define soma-digitos-de-positivo
  (lambda (num)
    (cond ((< num 10) num)
          (else (+ (digito-menos-significativo num)
                   (soma-digitos-de-positivo
                     (retira-digito-menos-significativo num)))))))
```

Tendo por base *soma-digitos-de-positivo*, escreva o procedimento *soma-digitos-melhorado* que não repete o teste *negative?* como acontecia em *soma-digitos*.

```
> (soma-digitos-melhorado 823)
13
> (soma-digitos-melhorado -823)
Numero negativo!!!
> (soma-digitos-melhorado 1234567890)
45
>
```

Espaço para desenvolver e experimentar o procedimento *soma-digitos-melhorado*. O ecrã abre os procedimentos *soma-digitos-de-positivo*, *digito-menos-significativo* e *retira-digito-menos-significativo*.

Exercício 3

Escreva em Scheme o procedimento *soma-n-digitos*, com os parâmetros *n* e *nd* que são números inteiros positivos, e que devolve a soma nos *nd* dígitos menos significativos de *n*.

Apresente o diagrama de fluxo de dados que relacione os vários procedimentos que venha a utilizar. Encare os exemplos que se seguem como um potencial conjunto de situações de teste e interprete cuidadosamente os respectivos resultados.

```
> (soma-n-digitos 123 2)
5
> (soma-n-digitos 123 3)
6
> (soma-n-digitos 123 50)
6
> (soma-n-digitos 123 0)
0
```

Pista
Recorra a um procedimento auxiliar *nd-menos-significativos* com os parâmetros *numero* e *nd*, que devolve o valor correspondente ao número composto pelos *nd* dígitos menos significativos de *numero*. Depois, bastará chamar *soma-digitos-melhorado* do exercício anterior.

```
> (nd-menos-significativos 123 2)
23
> (nd-menos-significativos 123 3)
123
> (nd-menos-significativos 123 50)
123
> (nd-menos-significativos 123 0)
0
```

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

Exercício 4
O procedimento *soma-digitos-mais-significativos* aceita dois argumentos *n* e *nd*, e devolve a soma dos *nd* dígitos decimais mais significativos do número *n*. Escreva em Scheme o procedimento *soma-digitos-mais-significativos*.

Pista
Comece por escrever o procedimento *n-digitos* que determina o número de dígitos de um número *n*. Por exemplo, `(n-digitos 4100)` devolve 4. Para além desta pista, verifique que o dígito mais significativo de *n* pode ser determinado através de `(quotient n (expt 10 (- (n-digitos n) 1)))`.

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

Exercício 5 - Este não é muito simples, pois dá que pensar
O procedimento *a-subir* tem apenas um parâmetro, *num*, que é um número inteiro positivo.
Este procedimento analisa os dígitos de *num*, do menos significativo para o mais significativo, e visualiza *sobe* se os dígitos se apresentarem em ordem ascendente ou *nao sobe* se não se verificar a ordem ascendente dos dígitos.
Escreva em Scheme o procedimento *a-subir* e apresente o diagrama de fluxo de dados que relacione os procedimentos utilizados.
Considere as seguintes situações de funcionamento do procedimento, que deverão ser posteriormente utilizadas em testes a realizar.

```
> (a-subir 97431)
sobe
```

```
> (a-subir 79431)
nao sobe
> (a-subir 10)
sobe
> (a-subir 11)
nao sobe
> (a-subir 0)
sobe
```

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

Pista
(para utilizar se não conseguir encontrar uma ideia para resolver este problema...)
Tente uma solução recursiva baseada na ideia de que a resposta será *sobe* se o dígito menos significativo é menor que o dígito imediatamente a seguir e ainda se todos os dígitos, retirado o menos significativo, estiverem em ordem crescente.