Capítulo 2- Recursividade

Procedimentos recursivos
Processos recursivos e iterativos
Recursividade linear e em árvore
Ordem de crescimento
    Linear, constante, exponencial e logarítmica
Procedimentos como Blocos ou Caixas-pretas

Exercícios e exemplos
    Procedimento potencia-melhorada (Ordem de crescimento logarítmica)
    Projecto - Caminho

Na definição de um procedimento podem aparecer chamadas a outros procedimentos, situação perfeitamente normal e fácil de entender. Basta pensar que ao procedimento chamador está associada uma tarefa que se decompõe em tarefas mais simples, estas associadas aos procedimentos chamados.
Já não será tão natural que um procedimento se chame a si próprio! Através de um raciocínio mais ligeiro, até poderá parecer que este procedimento nunca mais terminaria... A ideia, à partida, sem sentido, constitui uma das características mais interessantes e intrigantes da programação. No caso do *Scheme*, é mesmo uma das suas características fundamentais, e sem a qual praticamente nada se faria. Falamos de *Recursividade*, utilizada na definição dos Procedimentos *Recursivos*, isto é, dos procedimentos que se chamam a si próprios.
O exemplo que tipicamente se apresenta para ilustrar o conceito da recursividade é a função *factorial*. Mas outros exemplos vão aparecer com frequência e a *Recursividade* acompanhar-nos-á constantemente, surpreendendo-nos com soluções quase sempre simples e elegantes. É recorrendo à *Recurividade* que o *Scheme* implementa as tarefas repetitivas, já que não possui, pelo menos à partida, estruturas de repetição (*for*, *while*, e outras), como existem noutras linguagens de programação. Atenção, portanto, pois será extremamente importante o domínio perfeito deste tema.

1- A Recursividade que nos rodeia

A *Recursividade* faz parte do nosso quotidiano, tão naturalmente, que quase não damos por ela. Se se pretende subir umas escadas, toma-se sem pensar, as seguintes acções:

⇒ Ao atingir o cimo das escadas, a tarefa de as subir está terminada;
⇒ Enquanto o cimo não for atingido
- Avançar um degrau
- e retomar a tarefa de subir as escadas, mas agora, tendo já avançado um dos degraus, a dimensão do problema aparece mais reduzida.

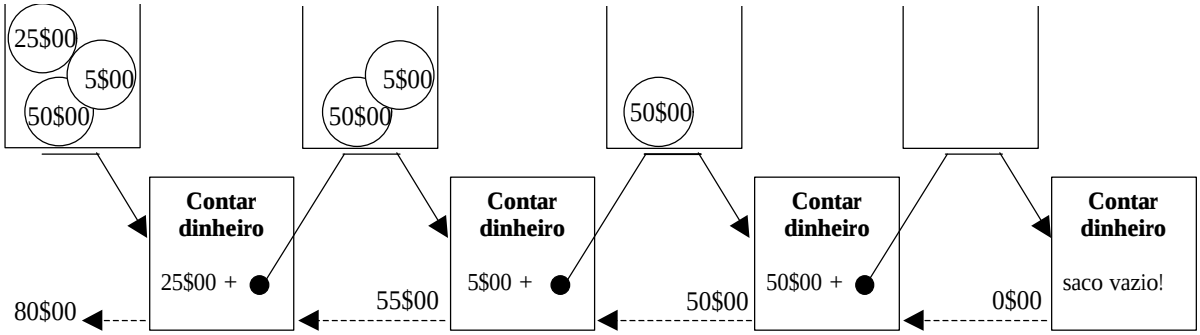
Identifica-se a *Recursividade* nestas acções quando, uma delas, retoma ou chama a tarefa inicial (subir as escadas). A garantia que a tarefa termina resulta de: 1- Na chamada seguinte o problema que se pretende resolver deve apresentar-se mais reduzido (já se subiu um degrau); 2- Deve existir uma condição de terminação (já se atingiu o cimo das escadas).

Considere-se agora um saco com moedas e pretende-se contar a quantia correspondente. Esta tarefa pode decompor-se nas seguintes acções:

⇒ Se o saco estiver vazio, sem moedas, isso corresponderá à quantia 0$00;

⇒ Enquanto o saco contiver moedas

- retira-se uma moeda do saco e identifica-se o seu valor
- que será adicionada à quantia correspondente às moedas que ainda restam no saco. Também aqui se retoma a tarefa inicial, quando se pretende saber a quantia correspondente às moedas que ainda estão no saco. Esta tarefa é retomada, mas com uma dimensão que é menor, devido à moeda que, entretanto, foi retirada.



Pode observar-se na figura que a contagem do dinheiro que se encontra no saco se inicia lançando o processo *Contar dinheiro*, representado pelo rectângulo mais à esquerda. Este processo de contagem identifica a quantia da moeda retirada, mas necessita também de conhecer a quantia 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. O processo situado mais à esquerda vai ficar suspenso até que lhe devolvam a informação pedida, ou seja, a quantia corresponde às moedas ainda no saco.

O segundo processo, pelas razões expostas para o primeiro, vai também lançar um novo processo, ficando também suspenso. E esta sequência de lançamento de novos processos continua até ao processo que detecta que o saco está vazio. Este é o único, até ao momento, que conhece a quantia que se encontra no saco, *0$00*, e por isso devolve-a ao processo que o lançou. Este processo não fica suspenso, pois resolveu completamente a tarefa que lhe foi solicitada. Entretanto, o processo suspenso que recebe a quantia de *0$00*, pode completar a sua tarefa devolvendo a quantia de *50$00*. Assiste-se agora ao finalizar dos processos suspensos. Quando o primeiro processo lançado recebe a quantia de *55$00*, já todos os outros processos desapareceram. E também este desaparecerá, logo após ter devolvido a quantia de 80$00, correspondente à quantia da totalidade das moedas contidas no saco.

Neste exemplo, a *Condição de terminação*, também designada por *Caso base*, equivale à situação de Saco vazio, a que corresponde uma quantia de *0$00*. Por seu turno, o *Passo recursivo*, também conhecido por *Caso geral*, traduz-se por retirar e avaliar uma moeda do saco, e chamar novamente a tarefa de contagem para as moedas restantes, dando assim um passo na direcção da *Condição de terminação.*

Exercício 2.1

Imaginar algumas situações da vida real que poderão ser resolvidas recorrendo ao conceito da Recursividade, identificando para cada uma delas a Condição de terminação e o Passo recursivo.

2- Procedimentos Recursivos

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

```
0! = 1 (por convenção)
1! = 1     = 1 * 0!
2! = 2 * 1 = 2 * 1!              = 2 * 1 = 2
3! = 3 * 2 * 1 = 3 * 2!          = 3 * 2 = 6
4! = 4 * 3 * 2 * 1 = 4 * 3!      = 4 * 6 = 24
...
n! = n * (n-1) * (n-2) * ... * 1  = n * (n - 1)!
```

Podemos daqui concluir que:

Caso base: `0! = 1` *; caso em que o valor é conhecido por definição ou convenção*
Caso geral `n! = n * (n-1)!` *; caso que recorre à própria função que se pretende resolver!...*

O *caso base* é conhecido por definição ou convenção, enquanto que o *caso geral* é definido recorrendo a uma chamada à própria função *factorial*. Note-se, contudo, que chamar *(n-1)!*, relativamente ao caso a resolver, *n!*, representa uma redução da complexidade do problema. Fala-se em redução, pois *(n-1)!* está mais perto do *caso base*, *0!*, que *n!*.
A partir desta definição de *factorial*, é possível escrever o respectivo procedimento.

```
(define factorial
  (lambda (num)
    (if (zero? num)

        1                ; o caso base

        (* num (factorial (sub1 num))))))    ; o caso geral
```

Para melhor se entender como funciona este procedimento recursivo, vamos seguir, passo a passo, o desenrolar de uma chamada concreta de *factorial*. Por exemplo,

```
↦(factorial 4)
24
```

Neste caso *num* = 4, portanto, diferente de zero. Seguindo a expressão *if* do procedimento *factorial*:

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

Um dos operandos da expressão composta, entretanto obtida, é (factorial 3), cujo cálculo implica nova chamada ao procedimento *factorial*. Agora, *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))))
```

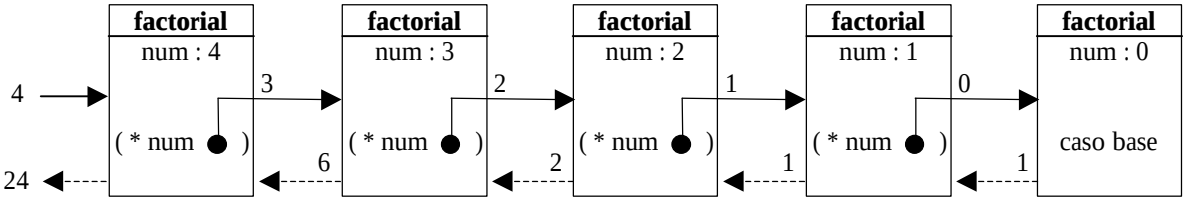
O comprimento da expressão vai aumentando, mas, em contrapartida, as chamadas ao procedimento *factorial* vão-se aproximando do *caso base*, o único de que se conhece a solução, por convenção. Continuando a seguir a chamada a *factorial* com *num* = 2, depois com *num* = 1 e, finalmente, com *num* =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

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

Neste ponto, atingiu-se o *caso base*, *0! = 1*, e a expressão vai começar a reduzir:

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

O funcionamento do procedimento recursivo *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*, vão sendo gerados processos todos eles para realizarem tarefas idênticas, mas de dimensão sucessivamente mais reduzida. 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 a resposta. A geração sequencial de novos processos continuará até se atingir o *caso base*, com *num = 0*, no processo representado pelo rectângulo mais à direita. 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:

⇒ Identificar a *condição de terminação* ou *caso base*, ou seja, o caso para o qual se conhece solução. No exemplo *factorial*, a condição de terminação é *num* = 0, para a qual *0! = 1*;

⇒ Identificar a operação de redução a aplicar, sucessivamente, até se atingir a *condição de terminação*. No mesmo exemplo, a operação de redução é *sub1* aplicada a *num*;

⇒ Escrever o procedimento recursivo, começando pelo *caso base*, passando seguidamente ao *caso geral*, de ordem *n*. Este é normalmente resolvido através de uma chamada recursiva ao procedimento que se pretende definir, chamada essa que deverá aproximar-se do *caso base*. Ainda no exemplo *factorial*, considerando a situação correspondente a *num*, a chamada recursiva passaria a ser sobre *(sub1 num)*.

Exemplo 2.1

O procedimento *soma-digitos* aceita um argumento inteiro e positivo, *numero*, e devolve a soma dos dígitos decimais do argumento.

```
↦(soma-digitos 123)
6
↦(soma-digitos -123)
Numero negativo!!!
↦(soma-digitos 000000)
0
```

Para definir este procedimento, a ideia a explorar resume-se a somar os dígitos do argumento, um a um, começando pelo menos significativo até atingir o mais significativo. Para aplicar a recursividade, toma-se o dígito menos significativo que será somado aos dígitos restantes, começando assim uma operação de redução do problema.

Por exemplo,

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

Neste exemplo, o problema inicial é de ordem 3, pois o número é composto por 3 dígitos, mas com esta definição passou a um problema de ordem 2, cuja solução é suposta conhecida. 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 esquecer que, neste exemplo, é necessário começar por verificar se o número é negativo e, se assim for, é apenas visualizada a mensagem correspondente.
Mas supondo que o número é positivo.

⇒ O *caso base* é: Número menor que 10, pois a resposta torna-se imediatamente conhecida. A soma dos dígitos é o próprio número;

⇒ A *operação de redução*: Operador *quotient* aplicado ao número (quociente da divisão inteira do número por 10) reduz a ordem do problema[1];

⇒ O *caso geral*: Somar o dígito menos significativo com a soma dos dígitos do número entretanto reduzido, soma, para o efeito, suposta conhecida.

```
(define soma-digitos
  (lambda (num)
    (if (negative? num)
        (display "Numero negativo!!!")
        (cond
           ((< num 10) num)            ; o caso base
           (else (+ (digito-menos-significativo num)  ; os outros casos
                    (soma-digitos (retira-dig-menos-signif num)))))))))

(define digito-menos-significativo
  (lambda (num)                        ; deste resto resulta
    (remainder num 10)))               ; o dígito menos significativo

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

É de salientar, nesta solução, dois aspectos. Um relaciona-se com a identificação e implementação das duas tarefas auxiliares em que se decompõe a tarefa *soma-digitos*, designadas por *digito-menos-significativo* e *retira-digito-menos-significativo*. O segundo aspecto, mais subtil, tem a ver com o teste *negative?*, repetido sucessivas vezes no procedimento *soma-digitos*, quando, na realidade, apenas seria necessário da primeira vez. Para resolver situações como esta, bastará

[1] Esta operação equivale a retirar o dígito menos significativo ao número.

pensar num procedimento auxiliar, *soma-digitos-aux*, que trata a solução para o caso ideal, em que o argumento nunca é negativo.

```
(define soma-digitos-aux
  (lambda (numero)
    (cond ((< numero 10) numero)
          (else (+ (digito-menos-significativo numero)
                   (soma-digitos-aux (quotient numero 10)))))))
```

Definido *soma-digitos-aux*, o procedimento *soma-digitos-melhorado* já não repete o teste *negative?* como acontecia em *soma-digitos*.

```
(define soma-digitos-melhorado
  (lambda (numero)
    (if (negative? numero)
        (display "Numero negativo!!!")
        (soma-digitos-aux numero))))
```

Exercício 2.2

Escrever 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*.

```
↦(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: Recorrer ao procedimento *soma-digitos-melhorado* e ao procedimento auxiliar *nd-menos-significativos*, que tem como parâmetros *n* e *nd*, e devolve o valor correspondente ao número composto pelos *nd* dígitos menos significativos de *n*.

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

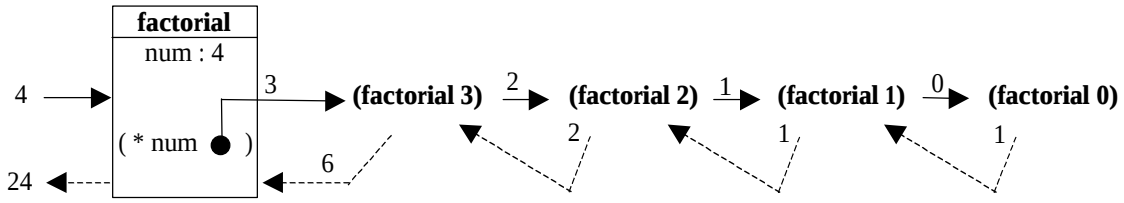
3- Processos recursivos e iterativos

Um procedimento recursivo é fácil de identificar, pois no seu corpo existirá uma chamada para si próprio, como se pode verificar no procedimento *factorial*.

```
(define factorial
  (lambda (num)
    (if (zero? num)
        1
        (* num (factorial (sub1 num))))))
```

Um procedimento, quando é chamado, executa a tarefa que lhe está associada. Ou seja, a chamada de um procedimento cria um *processo*, que gasta recursos materiais *(tempo de cálcul*o e

espaço de memória), mas que no final fornece o resultado da tarefa respectiva. Por exemplo, uma chamada do procedimento *factorial* cria um processo que toma um inteiro como argumento, processa-o e, no final, devolve o factorial desse valor. Sabemos, no entanto, que este processo desencadeia uma sequência de processos idênticos, até se atingir o *caso base*.




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

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

A chamada *(factorial 4)* cria um processo, que por sua vez cria outro processo, e assim sucessivamente até ao *caso base*, diferindo, ou seja, suspendendo a execução de uma cadeia de multiplicações, cadeia essa que vai progressivamente crescendo em memória. A partir do *caso base*, as multiplicações vão sendo calculadas, a expressão vai reduzindo de tamanho, e a memória vai sendo libertada. Podemos facilmente deduzir que os processos gerados pelas chamadas do procedimento *factorial* gastam recursos (tempo de cálculo e memória) proporcionais ao valor do argumento. Ou seja, o processo gerado pela chamada *(factorial 8)* deveria gastar aproximadamente o dobro do tempo e da memória que são gastos pelo processo gerado pela chamada *(factorial 4)*[2]. Esta característica, de diferir ou suspender a execução das operações, guardando-as em memória, até encontrar o *caso base*, caracteriza e identifica os chamados *Processos Recursivos*.

Convém aqui frisar a diferença entre *procedimento* e *processo*. O procedimento é o próprio texto, uma entidade estática que se pretende suficientemente legível, e que traduz uma ideia. O processo é uma entidade dinâmica, existe no interior do computador, gasta recursos, mas fornece resultados.

Encarando o cálculo do factorial de uma forma diferente, é possível definir outro procedimento, que cria um processo com outras características, mas que, no final, continua a fornecer o resultado pretendido, ou seja, *(factorial n)*.
Em vez de

```
n! = n * (n-1)!
```

encare-se antes o seguinte:

```
n! = (... ((n * (n-1)) * (n-2)) * ... * 1)
```

A ideia que daqui ressalta tem a ver com o facto das multiplicações serem calculadas a partir de *n* e *n-1*, resultado que é guardado, por exemplo, num *acumulador*, em vez de se suspender a respectiva execução. Seguidamente, o valor do *acumulador* será multiplicado por *n-2*, resultado

[2] Este tipo de previsão deverá verificar-se sobretudo para valores elevados de *n*, situação em que se tornam desprezáveis os recursos gastos de uma forma constante, e que nada têm a ver com a dimensão do problema

que vai actualizar o valor do *acumulador*. Este processo repete-se de passo em passo, ou de iteração em iteração, até que *n =0*, altura em que o valor de *(factorial n)* se encontra no *acumulador*.

Tomando o *(factorial 4)* como exemplo, *n* vai começar com o valor 4 e *acumulador* com o valor 1 *(valor neutro da multiplicação)*.

iteração	n	acumulador	acumulador *no final da iteração*
1ª	4	1	4
2ª	3	4	12
3ª	2	12	24
4ª	1	24	24
5ª	0 ------- *resultado* -> 24		

O *acumulador* é actualizado, no final de cada iteração, com o resultado do seu próprio valor multiplicado por *n*. Este *n* vai sendo reduzido de 1 em cada iteração, até se atingir o *caso base*, *n* = 0. Podemos agora escrever o procedimento *factorial-novo*, baseado na ideia acabada de expôr.

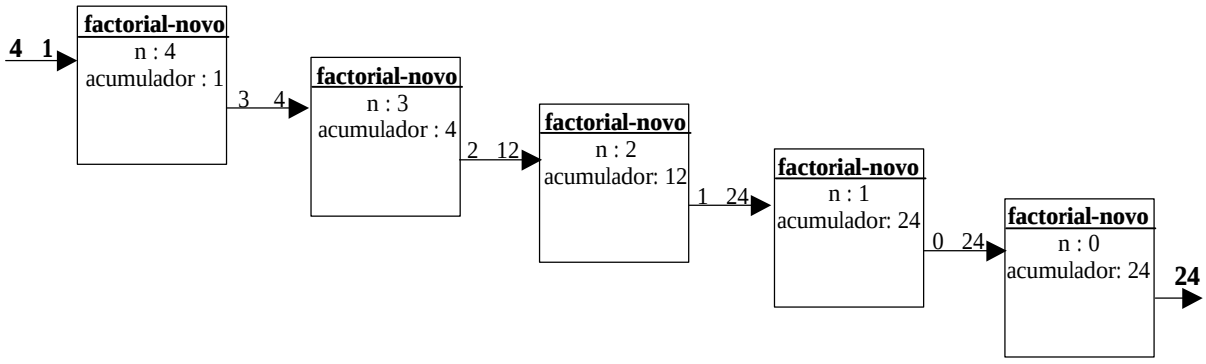
```
(define factorial-novo
   (lambda (n acumulador)
      (if (zero? n)

          acumulador              ; caso base

          (factorial-novo (sub1 n) (* n acumulador)))))     ; caso geral
```

```
↦(factorial-novo 4 1)
24
```

Vamos agora analisar, como se desenvolve, por exemplo, a chamada *(factorial-novo 4 1)*:

de `(factorial-novo 4 1)` resulta `(factorial-novo 3 (* 4 1))`
de `(factorial-novo 3 4)` resulta `(factorial-novo 2 (* 3 4))`
de `(factorial-novo 2 12)` resulta `(factorial-novo 1 (* 2 12))`
de `(factorial-novo 1 24)` resulta `(factorial-novo 0 (* 1 24))`
de `(factorial-novo 0 24)` resulta `24`

Tentando também uma representação gráfica para o mesmo exemplo, facilmente se observa que os sucessivos processos criados não suspendem a actividade, pois não ficam à espera de uma resposta. Limitam-se a lançar um processo idêntico a eles próprios, mas com outros argumentos, sempre na direcção do caso base.



O procedimento *factorial-novo* continua a ser um *procedimento recursivo*, pois reconhece-se no seu corpo uma chamada a si próprio. Verifique-se, contudo, que uma chamada deste procedimento, por exemplo, *(factorial-novo 4 1)*, cria um processo que não fica à espera de qualquer resposta. Este processo provoca a chamada *(factorial 3 4)*, e termina imediatamente a sua tarefa. Este modo de funcionamento continua com *(factorial 2 12)*, depois com *(factorial 1 24)* e, finalmente, com *(factorial 0 24)* que, por corresponder ao *caso base*, responde devolvendo o valor 24. Contrariamente ao que acontecia com *factorial*, *factorial-novo* vai executando as multiplicações à medida que aparecem, o que implica um gasto constante em memória (memória para guardar *num* e *acumulador*).
Podemos facilmente deduzir que os processos gerados pelas chamadas do procedimento *factorial-novo* gastam recursos constantes em memória e recursos em tempo de máquina proporcionais ao valor do argumento. Ou seja, o processo gerado pela chamada *(factorial-novo 8 1)* deve gastar, aproximadamente, o dobro do tempo e a mesma memória que são gastos pelo processo gerado pela chamada *(factorial-novo 4 1)*[3]. Estamos perante um *Processo Iterativo*, em geral, menos consumidor de recursos que o correspondente processo recursivo.

Não parece muito correcto fazer chamadas a procedimentos que calculam factoriais, fornecendo dois argumentos: o número de que se pretende conhecer o factorial e o valor com que se iniciliza o acumulador, normalmente o valor 1. Define-se o procedimento *factorial-iter*, baseado em *factorial-novo*, que esconde este inconveniente.

```
(define factorial-iter
   (lambda (num)
      (factorial-novo num 1)))
```

Analisando com atenção o procedimentos *factorial*, deduz-se que há lugar à *criação de processos recursivos* quando a chamada do procedimento recursivo não é o último passo do procedimento. De facto, senda a chamada recursiva um dos operandos de uma expressão e como os operandos são calculados em primeiro lugar, significa que a multiplicação terá que ficar suspensa até que se conheça o valor correspondente a *(factorial (sub1 n))*.

```
(* n (factorial (sub1 n)))   <-processo recursivo (em factorial)
```

Por outro lado, em procedimentos como *factorial-novo*, há lugar à *criação de processos iterativos*, pois a chamada do procedimento recursivo é o último passo do procedimento. Agora, a multiplicação encontra-se num dos argumentos da chamada recursiva e terá, por este motivo, de ser calculado em primeiro lugar. Só depois dos argumentos calculados é que a chamada terá lugar, sendo assim o último passo do procedimento.

```
(factorial-novo (sub1 n) (* n acumulador)))   <-processo iterativo
                                                    (em factorial-novo)
```

Esta recursividade, em que o passo recursivo se encontra na *cauda* do procedimento, é designada por *recursividade em cauda*[4].

Em termos de recursos de memória, os procedimentos recursivos que geram processos iterativos (*recursividade em cauda*, como se verifica em *factorial-novo*) são mais eficientes que os procedimentos recursivos que geram processos recursivos (como em *factorial*). Pelo menos, assim acontece em Scheme, cujo interpretador, segundo o standard do IEEE, deve apresentar

[3] Como já foi referido anteriormente, este tipo de previsão deverá verificar-se sobretudo para valores grandes de *n*, situação em que se tornam desprezáveis os recursos gastos de uma forma constante, e que nada têm a ver com a dimensão do problema.
[4] *Tail recursion*

uma implementação *tail-recursive*, em que um processo iterativo pode ser gerado por um procedimento recursivo. Noutras linguagens, os processos iterativos deverão ser gerados a partir de estruturas de repetição: *do*, *repeat*, *for*, *while*, e não através de procedimentos recursivos. Em termos de tempo de cálculo, os comportamentos podem ser semelhantes, ou menos exigentes, nos processos iterativos do que nos correspondentes processos recursivos. Todavia, a ideia que serve de base à definição dos procedimentos recursivos que geram processos recursivos (como *factorial*) é, normalmente, muito mais natural e simples de encontrar, do que a ideia necessária à definição dos procedimentos recursivos que geram processos iterativos (como *factorial-novo*).

Exemplo 2.2

Baseando-se numa solução recursiva, escrever um procedimento que toma um argumento n, inteiro e positivo, e gera um processo recursivo para calcular *n + (n-1) + ... + 1 + 0*

Neste exemplo, o *caso base* é *n = 0*, a *operação de redução* é *sub1* aplicada a *n*, e o *caso geral* é adicionar *n* à soma dos restantes números da série, suposta conhecida.

```
(define soma-sequencia
  (lambda (numero)
    (if (zero? numero)
        0
        (+ numero
           (soma-sequencia (sub1 numero))))))
```

```
↦(soma-sequencia 3)
6
```

Esta solução gera um processo recursivo, pois a chamada recursiva é um dos operandos da expressão, não sendo o último passo do procedimento. Isto significa que as operações vão sendo guardadas em memória, até que se encontre o caso base.
Tenta-se agora a escrita de um procedimento que executa a mesma tarefa, mas gerando um processo iterativo. Com este objectivo, introduz-se um acumulador para guardar o resultado das sucessivas adições, inicializado com zero, o elemento neutro da adição.

```
(define soma-sequencia-aux
  (lambda (numero acumulador)
    (if (zero? numero)
        acumulador
        (soma-sequencia-aux (sub1 numero)
                            (+ numero acumulador)))))

(define soma-sequencia-iter            ; procedimento para esconder
  (lambda (numero)                     ; os 2 argumentos
     (soma-sequencia-aux numero 0)))       ; aqui o acumulador começa em 0
```

Vamos agora testar as duas soluções, e tentar interpretar as respostas:

```
↦(soma-sequencia 3)
6
↦(soma-sequencia-iter 3)
6
↦(soma-sequencia 2000)
+: Out of stack space
```

Com um argumento elevado, *soma-sequencia* deixa em suspenso um número de processos recursivos que esgota a memória que o Scheme tem disponível.

```
↦(soma-sequencia-iter 2000)
```

```
2001000
```

Com o mesmo argumento, *soma-sequencia-iter* não tem problemas de falta de memória, pois gera processos iterativos que não suspendem a actividade e, por esse motivo, o espaço de memória necessário é constante.

```
↦(soma-sequencia 20000)
+: Out of stack space                    ; como era de esperar...

↦(soma-sequencia-iter 20000)
[Garbage collecting... 48K of 512K]      ; O Scheme reaproveita
[Garbage collecting... 48K of 512K]      ; automaticamente a memória
[Garbage collecting... 48K of 512K]      ; que vai sendo libertada
200010000
```

As mensagens que antecedem a resposta `200010000` apenas significam que, após a chamada de *soma-sequencia-iter*, o *Scheme* reaproveitou o espaço de memória que foi libertado pelos processos que, entretanto, terminaram. Este espaço de memória, classificado de *lixo*[5], libertado, por exemplo, pelos processos iterativos que não ficam suspensos como os recursivos, é reaproveitado automaticamente pelo Scheme através de uma operação de *recolha de lixo*[6].

Exercício 2.3

Escrever em *Scheme* o procedimento *soma-n-digitos-iter*, uma solução iterativa equivalente ao procedimento *soma-n-digitos* apresentado em Exercício 2.2.

Exemplo 2.4

Escrever em *Scheme* o procedimento *troca-posi*, que espera um único argumento, inteiro positivo, visualizando-o com os dígitos na ordem inversa, e com o caracter "." entre eles.

```
↦(troca-posi 123)
3.2.1

↦( troca-posi 5)
5
```

A solução que se apresenta baseia-se em dois procedimentos auxiliares, já utilizados no exemplo 2.1, *digito-menos-significativo* e *retira-digito-menos-significativo*. O caso base corresponde a um número menor que 10, representado por um único dígito. O caso geral traduz-se pela visualização do dígito menos significativo, e de um ".", finalizando como uma chamada recursiva a *troca-posi*, com um argumento equivalente ao actual, retirado o dígito menos significativo.

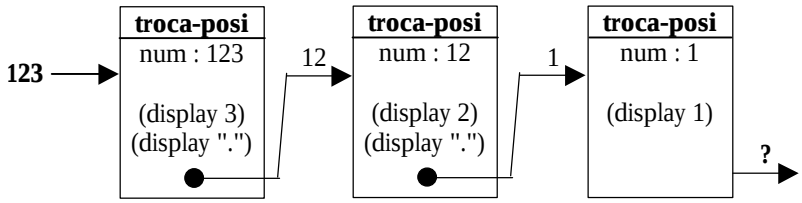
```
(define troca-posi
  (lambda (num)
    (cond
      ((< num 10)          ;caso base
       (display num))
      (else                                          ; caso geral
        (display (digito-menos-significativo num))
        (display ".")
        (troca-posi (retira-digito-menos-significativo num))))))
```

Neste procedimento, é utilizada *recursividade em cauda*, pois o último passo é o passo recursivo. Assim, o processo gerado é iterativo, o que também se comprova com a representação gráfica da chamada *(troca-posi 123)*.

---

[5] *garbage*

[6] *garbage collecting*



Também se verifica, pela representação gráfica, que o valor devolvido pelos processos não é importante, mas sim o efeito lateral correspondente à visualização no ecrã.

Exemplo 2.5

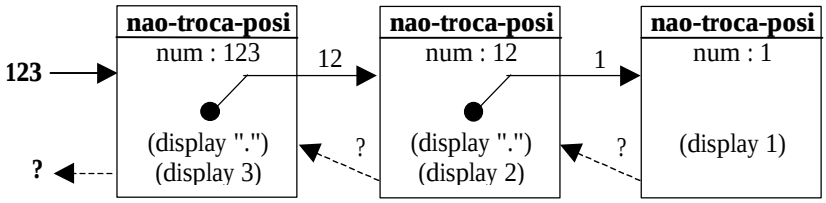
Escrever em *Scheme* o procedimento *nao-troca-posi*, que espera um único argumento, inteiro positivo, visualizando-o com os dígitos na ordem normal, e com o caracter "." entre eles.

```
↦(nao-troca-posi 123)
1.2.3
↦(nao-troca-posi 5)
5
```

A solução que se apresenta parece idêntica à utilizada para *troca-posi*, com umas trocas de posição de algumas instruções, mas uma análise mais cuidada leva a concluir que o processo agora gerado tem um comportamento muito diferente do anterior. Neste exemplo, não se utiliza *recursividade em cauda*. O passo recursivo, seguido por dois *display*, não é o último passo de *nao-troca-posi*, o que significa que os processos vão ficar suspensos até se atingir o caso base.

```
(define nao-troca-posi
  (lambda (num)
    (cond
      ((< num 10)
       (display num))
      (else (nao-troca-posi (retira-digito-menos-significativo num))
            (display ".")
            (display (digito-menos-significativo num))))))
```

Os processos gerados não são iterativos como no exemplo anterior, mas recursivos, conclusão que também se poderá retirar da representação gráfica relativa à chama *(nao-troca-posi 123)*.



Mais uma vez se verifica, pela representação gráfica, que o valor devolvido pelos processos não é importante, mas sim o efeito lateral correspondente à visualização no ecrã. A primeira visualização a acontecer situa-se no processo mais à direita, *(display 1)*.

4- Recursividade linear e em árvore

Nos procedimentos *factorial* e *factorial-novo*, soluções que criam, respectivamente, processos recursivos e iterativos, o tempo de máquina necessário crescia linearmente com o valor do número a processar. Estes procedimentos recursivos utilizam, por este motivo, *Recursividade linear*. A *Recursividade em árvore* apresenta outras características. O exemplo típico para ilustrar esta

recursividade é a sequência de Fibonacci, na qual, cada número é igual à soma dos dois números anteriores, sendo os dois primeiros, por convenção, 0 e 1.
Os primeiros membros da série são:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... que podem ser definidos a partir da regra

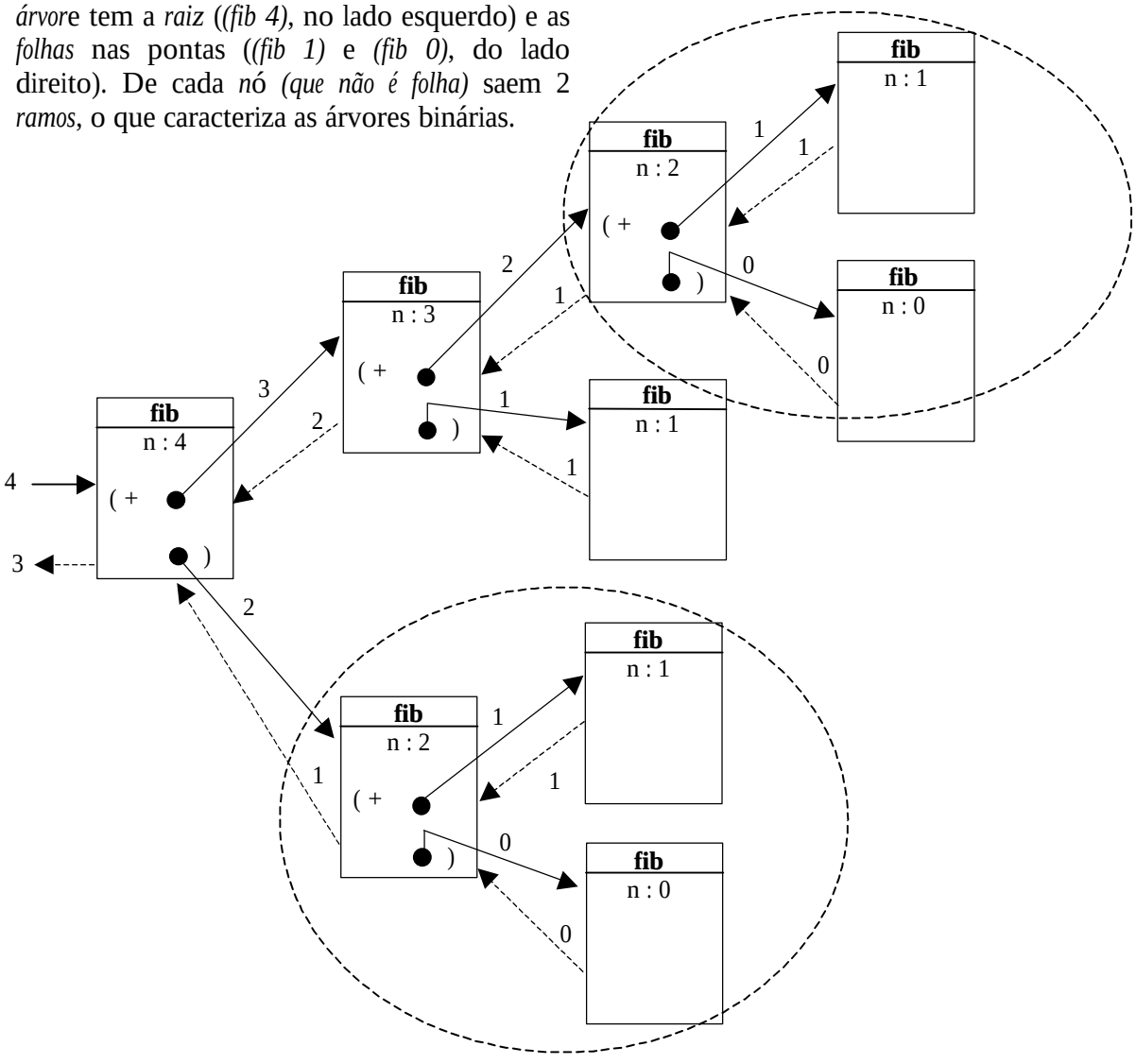
$$fib(n) = \begin{cases} n & \textit{se } n < 2 \\ fib(n-1) + fib(n-2) & \textit{nos outros casos} \end{cases}$$

Desta regra, retira-se facilmente o procedimento respectivo:

```
(define fib
   (lambda (n)
      (if (< n 2)
          n
          (+ (fib (- n 1))
             (fib (- n 2))))))
```

```
↦(fib 4)
3
↦(fib 11)
89
```

A chamada *(fib 4)*, sob a forma gráfica, mostra que o processo gerado é *recursivo em árvore*. A *árvor*e tem a *raiz* (*(fib 4)*, no lado esquerdo) e as *folhas* nas pontas (*(fib 1)* e *(fib 0)*, do lado direito). De cada *nó (que não é folha)* saem 2 *ramos*, o que caracteriza as árvores binárias.



Em termos de tempo, o comportamento do processo recursivo em árvore é terrivelmente gastador. Basta verificar na figura que, por exemplo, o cálculo *(fib 2)* é duplicado. O número de chamadas ao procedimento *fib* é igual ao número de nós da árvore, o que corresponde a um comportamento exponencial. Sendo *n* o dado inicial, o número de nós é q^n, em que *q* não chega a ser 2, apesar de saírem 2 ramos de cada nó[7], uma vez que a árvore não é perfeitamente simétrica. Em relação ao espaço de memória, verifica-se que o comportamento do processo é "apenas" linear, conclusão a que se chega seguindo a trajectória indicada pelas setas sobre a árvore. Os cálculos que se mostram nos diferentes ramos da árvore nunca estão suspensos ao mesmo tempo, pois vão sendo calculados à medida que se atinge algum dos casos base *(as folhas)*. Por exemplo, o ramo inferior, *(fib 2)*, só será percorrido depois de *(fib 3)* calculado[8]. O número máximo de cálculos suspensos, que é de 4, situa-se na zona da árvore com maior profundidade *(ramo onde se encontra (fib 3))*, profundidade que revela uma certa proporcionaldade linear com o número a processar.

O procedimento recursivo *fib* gera processos recursivos em árvore com um comportamento exponencial em tempo e linear em memória. Podemos sempre tentar um procedimento recursivo que gere um processo iterativo, bastante mais eficiente, quer em tempo *(linear)* quer em espaço de memória *(constante)*, tarefa que poderá revelar-se não muito simples. A ideia a explorar, para a solução iterativa, é manter dois acumuladores, aos quais se associam, em cada iteração, dois números seguidos da sequência de Fibonacci, o número corrente *(ac-corrente)* e o seguinte *(ac-seguinte)*. Serão inicializados com *ac-corrente* = fib(0) = 0, e *ac-seguinte* = fib(1) = 1.
Em cada iteração, desenrolar-se-ão, simultaneamente, as seguintes transformações:

- *ac-seguinte* vai para *ac-corrente*
- *ac-seguinte* + *ac-corrente* vai para *ac-seguinte*

Para completar a ideia, bastará inicializar um *contador* com *n*, o qual, em cada iteração, sofre uma redução de uma unidade. Quando *contador* atinge zero, a resposta encontra-se em *ac-corrente*.

```
(define fib-iter
  (lambda (contador ac-corrente ac-seguinte)
    (if (zero? contador)
        ac-corrente
        (fib-iter (sub1 contador)
                  ac-seguinte
                  (+ ac-seguinte ac-corrente)))))

(define fib-novo            ; procedimento para
  (lambda (n)               ; esconder os 2 dos parâmetros
    (fib-iter n 0 1)))      ; do procedimento fib-iter
```

Exemplo 2.6

Misturando cores, a partir de um conjunto de cores iniciais, produzem-se novas cores. Supõe-se, neste método de produção, que se utilizam sempre doses iguais de cada cor misturada. Assim, partindo das cores iniciais C1 e C2 conseguem-se as seguintes 3 cores distintas: C1, C2 e C1 + C2[9]. Com as cores iniciais C1, C2 e C3 já se conseguem 7 cores: C1, C2, C3, C1 + C2, C1 + C3, C2 + C3, e C1 + C2 + C3.

---

[7] Prova-se que *q*, designado por *golden ratio*, é igual a $q = (1+\sqrt{5})/2 \approx 1.6180$

[8] Como o *Scheme* não garante a ordem de cálculo dos operandos de uma expressão, poderá ser *(fib 3)* chamado só depois de *(fib 2)* calculado

[9] Aqui, o sinal + significa mistura de cores

Estamos interessados em saber quantas cores distintas é possível produzir a partir de 4, 5, ..., ou *n* cores iniciais. Com este objectivo

- Apresentar uma definição recursiva para determinar o número de cores distintas que se conseguem obter com *n* cores iniciais.
- Escrever em *Scheme* o procedimento *n-cores* com base na definição apresentada, que responda como se indica

```
↦(n-cores 0)
0
↦(n-cores 1)
1
↦(n-cores 2)
3
↦(n-cores 3)
7
↦(n-cores 4)
15
↦(n-cores 5)
31
↦(n-cores 10)
??
```

Resolução:

$$\text{nCores}(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ 1 + \text{nCores}(n-1) + \text{nCores}(n-1) & \text{nas outras situações} \end{cases}$$

Com n = 0 cores, não se produz qualquer cor. Com n = 1 cor, só se produz uma cor. Estas duas situações constituem o *Caso base*. Nas outras situações, consideremos uma cor qualquer do conjunto de conjuntos.

- Essa cor isolada produz 1 cor
- Sem essa cor, apenas com as restantes cores, produz-se nCores (n-1) distintas
- Agora, se a todas essas nCores (n-1) se juntar a cor que se isolou, obtém-se mais nCores (n-1) distintas.

```
(define n-cores
  (lambda (n)
    (if (< n 2)
        n              ; caso base
        (+ 1                          ; caso geral
           (* 2
              (n-cores (sub1 n)))))))
```

Exercício 2.4

Relativamente à situação do exemplo anterior, supõe-se agora que, por limitações tecnológicas, terá que haver um limite máximo de cores misturadas na obtenção de novas cores.

- Apresentar uma definição recursiva para determinar o número de cores distintas que se conseguem obter com *n* cores iniciais, sabendo que não se podem misturar mais do que k cores.
- Escrever em *Scheme* o procedimento *n-cores-lim* com base na definição apresentada, que responde como se mostra mais à frente.

- Apresentar a representação gráfica da chamada *(n-cores-lim 3 2).*
- Indicar se a recursividade utilizada é *linear* ou *em árvore*.

```
↦(n-cores-lim 3 0)
0
↦(n-cores-lim 3 1)
3
↦(n-cores-lim 3 2)
6
↦(n-cores-lim 3 3)
7
↦(n-cores-lim 3 4)
7
```

5- Ordem de crescimento

Os processos criados pelas chamadas de procedimentos consomem recursos computacionais, de uma forma muito diversificada. São os *recursos de tempo* de cálculo e os *recursos de espaço* de memória. Uma medida não muito precisa destes recursos, mas suficientemente útil, baseia-se na designada *Ordem de Crescimento*. Através da Ordem de Crescimento é possível caracterizar o consumo dos processos em função da "dimensão" dos dados que processam. Por exemplo, se a dimensão do problema duplica *(no caso do factorial seria: Se o número de que se pretende conhecer o factorial duplica...)* como é que este facto se manifesta nos recursos computacionais consumidos? Duplicam? São constantes? Aumentam exponencialmente? Ou aumentam logaritmicamente? Não se pretende uma medida muito rigorosa, mas apenas uma ideia da forma como reagirá um processo gerado por um procedimento, quando se faz variar os dados fornecidos.
Vamos considerar:

⇒ *n* o parâmetro que reflecte a dimensão do problema *(e que poderá ser, por exemplo, o valor numérico a processar, o número de células de um vector ou de uma matriz)*;

⇒ *R(n)* os recursos consumidos por um processo, para um *n* suficientemente elevado, que "dilua" os recursos constantes que não têm a ver com a dimensão do problema;

⇒ *O(f(n))* como a Ordem de crescimento de função *f(n)*, sendo *f(n)* constante, linear, ...;

⇒ Então, os recursos têm uma Ordem de Crescimento *f(n)*, ou seja, *R(n) = O(f(n))*, se, para um *n* suficientemente elevado, *R(n)* *K * f(n)*, sendo K um valor constante.

Vejamos alguns casos:

⇒ Se *R(n)* tiver um comportamento constante, independente de *n*, a Ordem de Crescimento diz-se *constante* e representa-se por *O(1)*. Por exemplo, R(n) = 1000 K * 1, f(n) = 1 e *O(1)*.

Aqui poderão começar a surgir algumas dúvidas, pois dois processos seriam ambos classificados com *O(1)*, apesar de um deles gastar constantemente recursos equivalentes a 1 unidade e outro processo gastar também constantemente recursos equivalentes a 1000 unidades!... Em termos absolutos, de facto, um dos processos gasta 1000 vezes mais recursos. Todavia, os dois processos reagem da mesma maneira, ou seja, consumindo recursos de uma forma constante, independente da dimensão do problema.

⇒ Se *R(n)* tiver um crescimento linear com *n*, a Ordem de Crescimento diz-se *linear* e representa-se por *O(n)*.

Aplicando o que se acaba de apresentar às soluções recursiva e iterativa do cálculo do factorial, podemos resumir.

- *Factorial*, com processos recursivos: Tempo: *O(n)*; Espaço: *O(n)*
- *Factorial-novo*, com processos iterativos: Tempo: *O(n)*; Espaço: *O(1)*.

⇒ Se *R(n)* tiver um crescimento K^n com *n*, a Ordem de Crescimento diz-se *exponencial* e representa-se por $O(K^n)$.
Sendo R(n) = 2^n, então $O(2^n)$, caso em que os recursos duplicam quando *n* aumenta de um valor unitário. Por exemplo, R(4) = 2^4 = 16, R(5) = 2^5 = 32, R(6) = 2^6 = 64, ...

Relembrando os procedimento *fib* e *fib-iter* da série de Fibonacci:

- *fib*: Tempo: $O(q^n)$, com q 1.6180 *(golden ratio*); Espaço: *O(n)*
- *fib-iter:* Tempo: *O(n);* Espaço: *O(1*).

⇒ Se *R(n)* tiver um crescimento $log_K n$ com *n*, a Ordem de Crescimento diz-se *logarítmic*a e representa-se por $O(log_K n)$.
Sendo R(n) = $log_2 n$, então $O(log_2 n)$, caso em que os recursos aumentam de um valor unitário quando *n* duplica. Por exemplo, se R(8) = $log_2 8$ = 3, R(16) = $log_2 16$ = 4, R(32) = $log_2 32$ = 5, ...

Trata-se de uma situação em que a Ordem de crescimento é melhor que a *linear*, muito melhor que a *exponencial* e apenas inferior à *constante*. Para já, não temos qualquer exemplo para apresentar em que se verifique alguma forma de Ordem de Crescimento *logarítmica*[10].

Como foi dito, a Ordem de Crescimento só nos permite caracterizar, de forma aproximada, o consumo de recursos em função da dimensão do problema. É por este motivo que se, por exemplo, R(n) = 12 n^7 + 30 n^6 + 13 n^3 + 100, classificaríamos a ordem de crescimento de $O(n^7)$. O erro que se comete desprezando os outros termos, mesmo para um *n=50*, não muito elevado, seria de, aproximadamente, 5%[11]. Para *n=100*, o erro já cairia para 2.5%.

6- Procedimentos como Blocos

Quando se referem os procedimentos como *Blocos* ou *Caixas-Pretas*, pretende-se pôr em destaque que estas entidades podem ser utilizadas conhecendo-se apenas *o que fazem* e não *como fazem*. O conhecimento do que se passa no interior de cada um desses blocos é dispensável. De facto, o que interessa conhecer sobre um procedimento para o utilizar é a sua *interface*, ou seja, a sua função e os argumentos que espera.

Com a forma especial *let*, apresentada no capítulo anterior, é possível definir no corpo dos procedimentos, outros nomes locais para além dos parâmetros. Esta possibilidade foi ilustrada, por exemplo, no capítulo anterior, quando se apresentou o procedimento *percentagem-lado-perimetro-2*, que é agora aqui retomado.

No que se segue, apresenta-se o campo de acção dos parâmetros *a*, *b*, *c*, e da variável local *perimetro*, definida por *let*.

```
(define percentagem-lado-perimetro-2
  (lambda (a b c)
    (let ((perimetro (+ a b c)))
      (display "perimetro do triangulo: ")
      (display perimetro)
      (newline)
      (display (* 100 (/ a perimetro)))
      (newline)
      (display (* 100 (/ b perimetro)))
      (newline)
      (display (* 100 (/ c perimetro))))))
```

a, b, c

perimetro

[10] Ver, no final do capítulo, o exemplo que apresenta várias implementações para a função *potencia*

[11] Considerando apenas o termo de maior grau, R(n) = $12n^7$, mas considerando os 2 de maior grau, R(n) = $12n^7$ + $30n^6$. Então, o erro relativo seria ($12n^7$ + $30n^6$ - $12n^7$) / $12n^7$ ou seja, para n = 50, 5% e, para n = 100, 2.5%

O que se pretende mostrar é que os parâmetros *a*, *b*, e *c* são apenas reconhecidos e acessíveis no corpo do procedimento *percentagem-lado-perimetro*, corpo que também inclui *let*. Quanto a *perimetro*, o seu campo de acção é mais limitado, reduzindo-se apenas ao corpo de *let*. Isto significa que os nomes acabados de referir, se utilizados fora destes campos de acção, representarão outras entidades, independentes destas.

```
↦(percentagem-lado-perimetro-2 2 3 4)
perimetro do triangulo: 9
22.2222222222222
33.3333333333333
44.4444444444444
```

Neste procedimento identifica-se uma tarefa que se repete três vezes, que visualiza a informação de cada um dos três lados do triângulo. Aproveitando este facto, apresenta-se uma nova implementação que utiliza o procedimento auxiliar *visu-lado*.

```
(define perc-lado-perimetro-1
  (lambda (a b c)
    (let ((perimetro (+ a b c)))
      (display "perimetro do triangulo: ")
      (display perimetro)
      (visu-lado a perimetro)
      (visu-lado b perimetro)
      (visu-lado c perimetro))))
(define visu-lado
  (lambda (lado perim)
    (newline)
    (display (* 100 (/ lado perim)))))
```

Mas não se fica pela implementação *perc-lado-perimetro-1*. Para além do nome de variáveis, é ainda possível definir com *let* procedimentos não recursivos no corpo de outros procedimentos, os quais terão também um campo de acção limitado ao corpo do *let* onde forem definidos. O procedimento *perc-lado-perimetro-2* define localmente o procedimento *visu* que substitui *visu-lado*.

```
(define perc-lado-perimetro-2
  (lambda (a b c)
    (let ((perimetro (+ a b c)))
      (let ((visu
              (lambda (lado)
                (newline)
                (display (* 100 (/ lado perimetro))))))
        (display "perimetro do triangulo: ")
        (display perimetro)
        (visu a)
        (visu b)
        (visu c))))
```


Uma vantagem desta solução é o nome *visu* só ser reconhecido no corpo de *let*, podendo ser utilizado fora deste ambiente, para designar outra entidade qualquer. Esta vantagem também pode ser vista como uma desvantagem, caso o procedimento *visu* fosse necessário a outros procedimentos, pois deixaria de estar acessível.

Outra vantagem deste tipo de construção foi também aproveitada em *perc-lado-perimetro*-2. Explorando o facto de *perimetro* ser acessível no corpo de *visu*, não foi necessário passá-lo como argumento, como acontecia em *visu-lado*. Assim, a chamada d*e visu* torna-se menos exigente em termos de recursos computacionais, portanto, mais eficiente, pois utiliza um menor número de argumentos. No corpo do procedimento *visu*, quando se utiliza *perimetro*, como não é conhecido localmente, recorre-se ao ambiente imediatamente acima. Diz-se, por este motivo, que este

nome não está limitado no procedimento *visu* e que é um nome *livre*, neste ambiente. Se no ambiente acima, *perimetro* continuasse a ser um nome livre (o que, de facto, não acontece), a procura continuaria até ao Ambiente Global do *Scheme* e a mensagem do tipo *top-level: unbound variable: perimetro* surgiria, pois a procura manifestar-se-ia infrutífera.
A definição de procedimentos dentro de outros procedimentos, através da forma especial *let*, reduz-se a procedimentos não recursivos. Com *letrec*, esta limitação é ultrapassada.
Relembrando a forma especial *let*, apresentada no capítulo anterior:

```
(let ( (nome-1 expressao-1)
       (nome-2 expressão-2)
       ( ...              ) )

   corpo-de-let)
```

Os símbolos *nome-1*, *nome-2*, ..., são apenas reconhecidos no corpo de *let*, e são associados aos valores das expressões respectivas. A forma especial *letrec* é muito parecida com a forma *let*.

```
(letrec ( (nome-1 expressao-1)
          (nome-2 expressão-2)
          ( ...              ) )

     corpo-de-letrec)
```

A forma *letrec*, também como acontece com *let*, devolve o valor da última expressão situada no seu corpo. Todavia, nesta forma, os símbolos podem ser associados a procedimentos recursivos ou mutuamente recursivos, significando que são reconhecidos não só no corpo de *letrec*, mas também no próprio espaço de definição desses símbolos.

Exemplo 2.7

Para ilustrar a utilização de *letrec*, retoma-se um exemplo já apresentado neste capítulo, onde são definidos o procedimento *soma-sequencia-iter* e o procedimento auxiliar *soma-sequencia-aux*, ambos no Ambiente Global do *Scheme*.

```
(define soma-sequencia-aux
  (lambda (numero acumulador)
    (if (zero? numero)
        acumulador
        (soma-sequencia-aux (sub1 numero)
                            (+ numero acumulador)))))

(define soma-sequencia-iter      ; procedimento para esconder
  (lambda (numero)                  ; os 2 argumentos
    (soma-sequencia-aux numero 0)))   ; aqui o acumulador começa em 0
```

Agora, através de *letrec*, apenas o procedimento *soma-sequencia-iter-1* é definido no Ambiente Global, enquanto que o procedimento auxiliar é definido no seu interior. Foi por este motivo que nem sequer houve uma grande preocupação na escolha do nome deste procedimento, *aux*, uma vez que se encontra muito bem delimitado o seu raio de acção.

```
(define soma-sequencia-iter-1
  (lambda (numero)
    (letrec ((aux
               (lambda (num acumulador)
                 (if (zero? num)
                     acumulador
                     (aux (sub1 num)(+ num acumulador))))))
      (aux numero 0))))
```

```
↦(soma-sequencia-iter-1 4)
10
```

Exercício 2.5

O procedimento *piramide* tem um único parâmetro, *base*, que deverá ser um inteiro positivo ímpar, e que representa o comprimento da base de uma "pirâmide".

```
↦(piramide 5)

  *
 ***
*****


↦(piramide 6)
Nao e' impar!!!
```

Escrever em *Scheme* o procedimento *piramide*, definindo os procedimentos auxiliares no interior dele, e classificar a solução encontrada em termos de Ordem de crescimento.

Exemplo 2.8

Dois procedimentos mutuamente recursivos[12] determinam se um número, *num*, é ou não ímpar[13], ou se é ou não par. Considerando o procedimento *numero-impar?*: O caso base é *num* = 0, correspondendo à resposta *#f*, pois zero não é ímpar. Os outros casos resumem-se a determinar se *num - 1* é par. De facto, a pergunta "*num é ímpar?*" deverá ter a mesma resposta que a pergunta "*num - 1 é par?*". A questão colocada desta forma, reduz o problema, na direcção do caso base.

```
(define numero-impar?
  (lambda (numero)
    (if (zero? numero)
        #f
        (numero-par? (sub1 numero)))))
```

Por um raciocínio análogo, chegar-se-ia ao procedimento *numero-par?*.

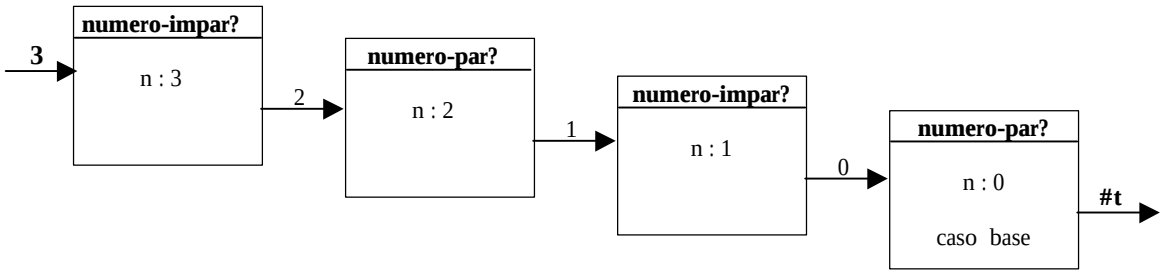
```
(define numero-par?
  (lambda (numero)
    (if (zero? numero)
        #t
        (numero-impar? (sub1 numero)))))
```

```
↦(numero-impar? 3)
#t
↦(numero-par? 3)
#f
```

E agora, através da chamada *(numero-par? 3)*, sob a forma gráfica, pode verificar-se que estes dois procedimentos, mutuamente recursivos, geram processos iterativos (nenhum deles fica

[12] Um procedimento chama outro procedimento que, por sua vez, chama o procedimento que o chamou
[13] Notar que o Scheme disponibiliza os procedimentos primitivos *odd?* e *even?*, predicados que indicam se o argumento é ímpar ou par, respectivamente (Anexo A)

suspenso, à espera de qualquer resposta) e apresentam *O(n)* em termos de tempo e *O(1)* em relação à memória utilizada.



Segue-se o procedimento *e'-impar?* que utiliza dois procedimentos auxiliares, mutuamente recursivos, definidos localmente com *letrec*.

```
(define e'-impar?
  (lambda (numero)

    (letrec ((numero-impar?
               (lambda (num)
                 (if (zero? num)
                     #f
                     (numero-par? (sub1 num)))))

             (numero-par?
               (lambda (num)
                 (if (zero? num)
                     #t
                     (numero-impar? (sub1 num))))))

      (numero-impar? numero))))
```

```
↦(e'-impar? 40)
#f
↦(e'-impar? 41)
#t
```

Exercícios e exemplos de final de capítulo

Nesta Secção, são apresentados alguns exercícios e exemplos para consolidação da matéria dada neste Capítulo sobre *Recursividade*. Recomenda-se o estudo e a resolução de todos eles, em frente do computador, uma vez que esta matéria vai ser fundamental, no decurso dos nossos trabalhos.

Exemplo 2.9

Uma versão simplificada deste exercício já foi considerada no Capítulo anterior: Uma bola é largada de uma altura *h* sobre uma superfície lisa, ficando a saltar durante algum tempo. Supor que ao saltar a bola toca a superfície sempre no mesmo ponto. A distância percorrida pela bola é a soma dos movimentos descendentes e ascendentes. Em cada salto, a bola sobe a uma altura que é dada pelo produto da altura do salto anterior por um factor *r* $(0 < r < 1)$, designado por *coeficiente de amortecimento*.

Escrever o procedimento *distancia-n*, que toma os valores *h*, *r*, e *n*, e devolve a distância percorrida pela bola desde o momento que é largada da altura *h* até ao final de *n* saltos, ou seja, após *n* descidas e *n* subidas.

No capítulo anterior apenas foram considerados os casos relativos a um e dois saltos, tendo sido definidos os procedimentos *distancia-1* e *distancia-2*.

```
↦(distancia-1 2 .5)
3.0
↦(distancia-1 3 .2)
3.6
↦(distancia-2 2 .5)
4.5
↦(distancia-2 3 .2)
4.32
```

E agora com *distancia-n*:

```
↦(distancia-n 3 .2 2)
4.32
↦(distancia-n 3 .2 20)
4.49999999999995
```

Resolução

Se se considerar que um salto é composto por uma descida e por uma subida, podemos estabelecer o seguinte:

Numero de saltos = 0, trata-se do caso base e resposta = 0.

Numero de saltos > 0, trata-se do caso geral e resposta = descida + subida + saltos restantes.

```
(define distancia-n
  (lambda (h r n-saltos)
    (let ((salto-temp (salto h r)))
      (if (zero? n-saltos)
          0
          (+ h                  ; descida
             salto-temp             ; subida
             (distancia-n salto-temp r (sub1 n-saltos))))))) ; restantes

(define salto
  (lambda (h r)
    (* h r)))
```

Nesta solução, introduziu-se a variável local *salto-temp* (salto temporário), para evitar uma repetição de cálculos.

```
↦(distancia-n 3 .2 200)
4.5
↦(distancia-n 3 .2 2000)
+: Out of stack space
↦(distancia-n 3 .2 700)
[Garbage collecting... 93K of 512K]
4.5
```

Exercício 2.6

Tendo por base o exemplo anterior:

- Explicar as respostas obtidas nas chamadas para 700 e 2000 saltos.
- Facilmente se identifica que a solução apresentada gera processos recursivos. Procurar uma solução iterativa para este problema, testando-a com números elevados de saltos.
- Identificar e justificar a Ordem de Crescimento das soluções recursiva e iterativa, em relação ao espaço e ao tempo.

Exercício 2.7

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*.
Escrever em *Scheme* o procedimento *soma-digitos-mais-significativos*.
Sugestão: Começar por escrever o procedimento *n-digitos* que determina o número de dígitos de um número n. Por ex: (n-digitos 4100) devolve 4. Para além desta sugestão, verificar que o dígito mais significativo de *n* pode ser determinado da seguinte maneira:

(quotient n (expt 10 (- (n-digitos n) 1))).

Exemplo 2.10

O procedimento *impar-cima-par-baixo* espera um argumento inteiro positivo. Se esse inteiro for par, dividi-o por 2. Se for ímpar, multiplica-o por 3 e depois soma 1. O resultado assim obtido é sujeito a um tratamento idêntico ao indicado e só pára quando o resultado for 1.

```
↦(impar-cima-par-baixo 6)
6  3  10  5  16  8  4  2  1  OK
```

Resolução

```
(define impar-cima-par-baixo
  (lambda (num)
    (display num)
    (display "  ")
    (cond ((= num 1)
           (display "OK")
           (newline))
          ((even? num)
           (impar-cima-par-baixo (quotient num 2)))
          (else
            (impar-cima-par-baixo (add1 (* num 3)))))))
```

Exercício 2.8

Em relação ao exemplo anterior, apresentar uma solução que prevê enganos do utilizador, quando indica um argumento que não obedece às condições enunciadas. Utilizar, para este efeito, um procedimento auxiliar não definido no Ambiente Global do *Scheme*.

Exemplo 2.11

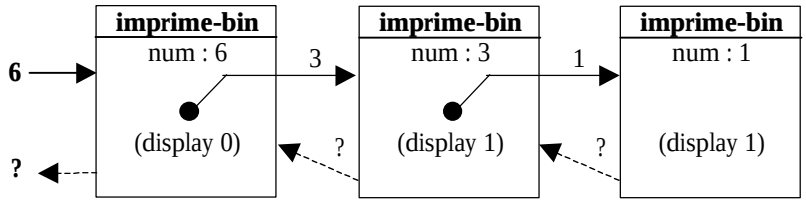
O procedimento *imprime-bin* espera um argumento inteiro positivo, na base 10. Este procedimento imprime o inteiro dado, convertido para a base 2.

```
↦(imprime-bin 76)
1001100
↦(imprime-bin 6)
110
```

Resolução

```
(define imprime-bin
  (lambda (num)
    (cond ((< num 2) (display num))
          (else
            (imprime-bin (quotient num 2))
            (display (remainder num 2))))))
```

Os processos gerados não são iterativos, mas sim recursivos, com *O(n)* em tempo e espaço, conclusão que também se poderá retirar da representação gráfica relativa à chama *(imprime-bin 6)*.



Verifica-se, pela representação gráfica, que o valor devolvido pelos processos não é importante, mas sim o efeito lateral correspondente à visualização no ecrã.

Variante
O procedimento *imprime-na-base* tem 2 parâmetros, *num* e *base*, para os quais espera dois argumentos inteiros positivos, ambos na base 10. Este procedimento imprime *num* na base especificada por *base*.

```
↦(imprime-na-base 76 2)
1001100
↦(imprime-na-base 76 3)
2211
↦(imprime-na-base 76 10)
76
```

Resolução

```
(define imprime-na-base
  (lambda (num base)
    (cond ((< num base) (display num))
          (else
            (imprime-na-base (quotient num base) base)
            (display (remainder num base))))))
```

Exercício 2.9

Observar o comportamento do procedimento *imprime-na-base*, quando o argumento correspondente à base é superior a 9.

```
↦(imprime-na-base 76 11)
610                          ; deveria ser 6a
↦(imprime-na-base 76 16)
412                          ; deveria ser 4c
```

Desenvolver um procedimento designado por *imprime-ate-base-20*, que responda correctamente, até à base 20, ou seja, para além dos dígitos decimais (0 a 9) ainda utiliza as letras de *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, e *i*.

```
↦(imprime-ate-base-20 76 11)
6a
↦(imprime-ate-base-20 76 16)
4a
↦(imprime-ate-base-20 76 20)
3g
```

Exemplo 2.12

O procedimento *caminho-errante* espera, como argumento, um valor inteiro que é considerado a origem, e gera os números inteiros 1, 2 e 3, numa sequência aleatória. Quando o número gerado

é 1 subtrai 1 à origem, quando é 2 soma-lhe 1, e quando é 3 nada altera. Este procedimento não tem fim.

```
↦(caminho-errante 20)
20  19  19  20  21  22  22  21  20  21  22  22  23  24  23  22  23
22  23  24  25  26  25  26  26  26  26  26  25  25  25  26
let: interrupt!
```

Resolução
Notar que neste procedimento recursivo não existe Caso base.

```
(define caminho-errante
  (lambda (origem)
    (display origem)
    (display "  ")
    (let ((numero-aleat (roleta 3)))

      (cond ((= numero-aleat 1)
             (caminho-errante (sub1 origem)))
            ((= numero-aleat 2)
             (caminho-errante (add1 origem)))
            (else
              (caminho-errante origem))))))

(define roleta              ; gera, aleatoriamente, valores
  (lambda (limite)                              ; entre 1 e limite
    (add1 (remainder (random) limite))))
```

Exercício 2.10
Escrever em *Scheme* o procedimento *caminho-errante-2d* que espera, como argumentos, dois valores inteiros, considerados as coordenadas *x* e *y* de um ponto 2D, tomado como origem. O procedimento gera os números inteiros 1, 2 e 3, numa sequência aleatória. Quando o número gerado é 1 subtrai 1 à coordenada *x*, quando é 2 soma 1 a essa mesma coordenada, e quando é 3 nada altera. Depois de gerar mais um número aleatório entre 1 e 3, faz a mesma coisa para a coordenada *y*. Este procedimento não tem fim.

```
↦(caminho-errante-2d 20 20)
20:20  21:21  22:20  21:21  21:21  21:20  21:20  22:20  22:19
22:18  21:18  21:19  20:19  21:20  22:20  22:19  21:19  22:20
21:19  20:20  21:21  20:21  19:21  19:22  18:21  19:22  18:22
19:23  19:24  18:24  19:24  18:24  17:25  18:25  18:24
let: interrupt!
```

Exemplo 2.13
Um dos objectivos deste exemplo é mostrar um caso em que a ordem de complexidade é logarítmica, $O(\log_k n)$.
Vamos esquecer por momentos que o *Scheme* disponibiliza o procedimento *expt*, cuja chamada (expt b n) devolve b^n. Pretende-se desenvolver o procedimento *potencia* que espera dois argumentos, *b* e *n*, sendo *n* um inteiro positivo, e a chamada (potencia b n) devolve b^n.

De imediato, surge uma definição recursiva:

- Caso base: $b^0 = 1$
- Caso geral: $b^n = b.b^{n-1}$

Desta definição resulta:

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

O processo gerado por *potencia* é recursivo e, em termos de tempo e espaço, apresenta ordens de crescimento *O(n)*.

Uma solução iterativa é obtida com um procedimento auxiliar que apresenta um terceiro parâmetro, que acumula o resultado das sucessivas multiplicações.

```
(define potencia-iter
  (lambda (b n)
    (letrec ((aux
               (lambda (b contador acumulador)
                 (if (zero? contador)
                     acumulador
                     (aux b (sub1 contador) (* b acumulador))))))

      (aux b n 1))))
```

O processo gerado por *potencia-iter* é iterativo e, em termos de tempo, é *O(n)* e, em termos de espaço, é *O(1)*.

Neste problema, é ainda possível explorar uma outra ideia, em que a redução do expoente acontece muito mais rapidamente.

- Caso base: $b^0 = 1$
- Casos gerais
  - Se n é par: $b^n = (b^{n/2})^2$
  - Se n é ímpar: $b^n = b.b^{n-1}$

```
(define potencia-melhorada
  (lambda (b n)
    (let ((quadrado
            (lambda (x)
              (* x x))))

      (cond ((zero? n) 1)
            ((even? n)(quadrado (potencia-melhorada b (/ n 2))))
            (else
              (* b (potencia-melhorada b (sub1 n))))))))
```

O processo gerado por *potencia-melhorada* é recursivo e, em termos de tempo e espaço, é $O(\log_2 n)$. Basta reflectir um pouco para se chegar a esta conclusão. Sempre que o expoente duplica, passando de *n* para *2n*, duplicando a dimensão do problema, apenas mais uma multiplicação é diferida.

Exercício 2.11
Procurar uma solução iterativa para *potencia-melhorada*, a designar por *potencia-melhorada-iter*, identificando e justificando a Ordem de Crescimento da solução encontrada.

Por tentativas, determinar a partir de que valor de expoente o procedimento *potencia-melhorada* esgota a memória disponível e verificar como se comporta *potencia-melhorada-iter* para esse valor de expoente.

Exercício 2.12

Desenvolver uma definição recursiva para determinar o número de hipóteses de trocar *n* escudos, assumindo a existência das moedas de 1, 5, 10, 20, 50, 100, e 200 escudos, e as notas de 500, 1000, 2000, 5000 e 10000 escudos.
Tendo por base a definição recursiva desenvolvida, escrever em *Scheme* o procedimento *num-trocas* com um único parâmetro, *quantia*, que representa uma quantia em escudos e devolve o número de trocas possíveis dessa quantia com as moedas e notas acima indicadas.
Escrever em *Scheme* o procedimento *lista-trocas* com o mesmo parâmetro *quantia*, mas que, em vez de devolver o número de trocas, visualiza todas as trocas possíveis.

Exercício 2.13

O procedimento *somas-iguais* tem dois parâmetros, *n* e *soma*, ambos inteiros positivos. A chamada *(soma-iguais 9 40)* determina e devolve o número de hipóteses distintas de adicionar números de 1 a 9, sendo 40 a soma 40 deles.

```
↦(soma-iguais 9 40)          ; 40 = 1+4+5+6+7+8+9
3                            ; 40 = 2+3+5+6+7+8+9
                             ; 40 = 1+2+3+4+6+7+8+9
```

- Apresentar uma definição recursiva para determinar o número de hipóteses distintas, de adicionar números de 1 a *n*, perfazendo *soma*.
- Escrever em *Scheme* o procedimento *somas-iguais* com base na definição apresentada.
- Indicar se a recursividade utilizada é *linear* ou *em árvore*.
- Indicar a Ordem de crescimento da solução apresentada, em termos de tempo e espaço.

Exercício 2.14

Escrever em *Scheme* o programa *numero-de-somas-iguais* com três parâmetros, *n*, *lim-inf* e *lim-sup* e que se baseia no procedimento do exercício anterior, *somas-iguais*. Este programa responde do seguinte modo:

```
↦(numero-de-somas-iguais 9 20 24)
Numeros de 1 a 9
20: a
21: b
22: c
23: d
24: e
```

em que *a*, *b*, *c*, *d*, e *e* representam, respectivamente, o número de somas iguais a 20, 21, 22, 23, e 24, conseguidas com os números de 1 a 9.

Exercício 2.15

Escrever em *Scheme* um programa designado por *teste-da-tabuada*, com um só parâmetro, *num*, e que põe *num* questões sobre a tabuada de multiplicar (tabuadas de 1 a 10). Os números que surgem nas questões são gerados aleatoriamente.

```
↦(teste-da-tabuada 10)   ; com esta chamada, serão postas 10 questões

3 x 8 = 24            ; Na primeira questão, o programa visualiza 3 x 8 e o
Resposta certa       ; utilizador escreveu 24. Portanto, resposta certa.

2 x 7 = 15
```

```
Resposta errada

...
                        ; No final das questões
Muito bem                   ; esta será a mensagem se o número de erros for inferior a 2
Deve estudar melhor a tabuada       ; ou esta, se o número de erros for 2 ou maior
```

Projecto 2.1 - Caminho

Vamos imaginar um tabuleiro com 25 células, organizadas numa matriz 5 x 5.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Numa das células do tabuleiro é colocado uma tartaruga que apenas se desloca na horizontal para a célula imediatamente ao lado *(nosso lado direito)* ou na vertical para a célula imediatamente abaixo.
Chegando à coluna 5, ou seja, a coluna mais à direita, a tartaruga não pode deslocar-se mais na horizontal. Também, quando atinge a linha 5, a linha do fundo, a tartaruga não pode deslocar-se mais na vertical. O objectivo da tartaruga é atingir a célula 25.
O caminho pode complicar-se, pois é possível colocar obstáculos intransponíveis em várias células. Por exemplo, colocando obstáculos nas células 2, 9, 12, 14, 15, 17 e 23, o tabuleiro toma o aspecto indicado na figura ao lado.

1	~~2~~	3	4	5
6	7	8	~~9~~	10
11	~~12~~	13	~~14~~	~~15~~
16	~~17~~	18	19	20
21	22	~~23~~	24	25

A tartaruga segue um caminho de acordo com uma estratégia muito simples: Desloca-se prioritariamente na horizontal e, só quando fica bloqueada neste movimento (ou encontrou um obstáculo ou alcançou a coluna 5) é que tenta deslocar-se na vertical.

1- Escrever em *Scheme* o procedimento *caminho*, que espera um argumento que é o ponto de partida, onde se coloca a tartaruga, e vai indicando o caminho percorrido por esta, de acordo com a estratégia indicada.

```
↦(caminho 1)
1; 6; 7; 8; 13; 18; 19; 20; 25; consegui
↦(caminho 11)
11; 16; 21; 22; falhei
↦(caminho 2)
partida errada
↦(caminho 3)
3; 4; 5; 10; falhei
```

Solução:
Representar o tabuleiro, com as características referidas, incluindo os obstáculos, constitui um dos problemas que convém, desde já, resolver. Infelizmente, não temos ainda conhecimentos sobre *Scheme* que nos permitam representar ou criar um modelo computacional do tabuleiro como um objecto único. Vamos ter que o considerar como um conjunto de 25 variáveis, designadas por c1, c2, ..., c25, uma por cada célula do tabuleiro, atribuindo-lhes, respectivamente, o valor *#t* ou *#f*, conforme tenham ou não obstáculo. Por agora, aceita-se tal solução.

```
; modelação do tabuleiro como se fosse um conjunto de 25 variáveis
;
(define c1 #f)     ; célula livre
(define c2 #t)     ; célula ocupada
(define c3 #f)     ; célula livre
...
```

```
(define c24 #f)   ; célula livre
(define c25 #f)   ; célula livre
```

O procedimento *caminho* verifica se o ponto de partida, fornecido como argumento, coincide com uma célula ocupada por um obstáculo. Se assim for, a mensagem correspondente é imediatamente visualizada. Tratando-se de uma célula livre, inicia-se a caminhada.

```
(define caminho
  (lambda (partida)
    (if (em-obstaculo? partida)
        (display "partida errada")
        (avancar partida))))
```

O procedimento *caminho* utiliza o procedimento *em-obstaculo?*, cuja implementação, apresentada mais à frente, reflecte bem a inadequada modelação escolhida para o tabuleiro. Este assunto será retomado, noutro capítulo, quando se tratar do tema *Abstracção de Dados*.
O procedimento *avancar* é o ponto central deste problema. Trata-se de um procedimento recursivo, com dois casos de terminação ou casos base:

⇒ Quando *posicao* (variável que modela a posição da tartaruga) atinge a célula 25;

⇒ Quando a tartaruga fica bloqueada (não consegue deslocar-se nem na horizontal nem na vertical).

O procedimento *avancar* apresenta dois casos gerais, com os quais reduz a dimensão do problema, pois ambos recorrem a chamadas recursivas de *avancar*, mas para uma dimensão do problema mais reduzida, ou seja, vão ambos na direcção dos casos base. São os seguintes, os casos gerais:

⇒ Estando a tartaruga bloqueada na horizontal, avança para a linha seguinte:

```
(avancar (+ 5 posicao))  ; avançar para a célula imediatamente abaixo, na linha
                         ; seguinte, corresponde a somar 5 à posição
                         ; actual da tartaruga
                         ; Ter em conta a identificação das células do tabuleiro
```

⇒ Não estando bloqueada na horizontal, avança sobre essa linha:

```
(avancar (add1 posicao)) ; avançar para a célula imediatamente ao lado, na
                         ; mesma linha corresponde a somar 1
                         ; à posição actual da tartaruga
```

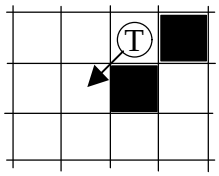
⇒ A operação de redução traduz-se em adicionar um valor a *posicao*, tentando assim chegar a 25. Num dos caso, adicionando-lhe 1 *(movimento na horizontal)*, no outro caso, adicionando-lhe 5 *(movimento na vertical)*.

```
(define avancar
  (lambda (posicao)
    (display posicao)
    (display "; ")
    (cond
      ((= posicao 25)          ; atingiu a célula 25
       (display "consegui")
       (newline))
      ((bloqueado? posicao)
       (display "falhei")      ; ficou bloqueado
       (newline))
      ((bloqueado-horiz? posicao)   ; bloqueado na horizontal?
       (avancar (+ 5 posicao)))      ; Sim. Avança na vertical
      (else
        (avancar (add1 posicao))))))) ; Não. Avança na horizontal

(define bloqueado?         ; bloqueado significa...
  (lambda (posicao)
    (and (bloqueado-horiz? posicao)    ; bloqueado na horizontal
         (bloqueado-vert? posicao))))  ; e bloqueado na vertical
```

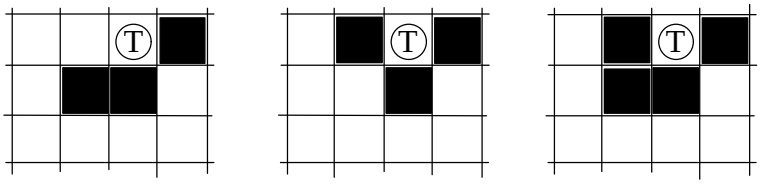
```
(define bloqueado-horiz?  ; bloqueado na horizontal significa estar...
  (lambda (posicao)
    (or
      (em-coluna-direita? posicao)        ; na coluna 5
      (em-obstaculo? (add1 posicao)))))  ; ou ter um obstáculo ao lado

(define bloqueado-vert?   ; bloqueado na vertical significa estar...
  (lambda (posicao)
    (or
      (em-linha-fundo? posicao)         ; na linha 5, ou ter um
      (em-obstaculo? (+ posicao 5))))) ; obstáculo por baixo

(define em-coluna-direita?   ; está encostado à direita?
  (lambda (posicao)
    (zero? (remainder posicao 5))))

(define em-linha-fundo?   ; está encostado ao fundo?
  (lambda (posicao)
    (> posicao 20)))

(define em-obstaculo?  ; devolve o valor da célula onde
  (lambda (posicao)    ; a tartaruga está posicionada
    (cond
      ((= 1 posicao) c1)
      ((= 2 posicao) c2)
      ((= 3 posicao) c3)
      ...
      ((= 24 posicao) c24)
      ((= 25 posicao) c25))))
```

2- Com base na solução apresentada, escrever em *Scheme* uma nova versão do procedimento *caminho*, agora com a designação de *caminho-v2*, em que a tartaruga, na situação de bloqueda na horizontal e na vertical, ainda tenta em terceira prioridade, um deslocamento na diagonal, conforme se indica na figura.

Situação em que ainda é possível mais uma tentativa:



Situações em que não é possível outra tentativa:



Colocando um obstáculo em *c8*, verificar o que se obtém quando se chama *caminho* e *caminho-v2*:

```
↦(caminho 1)
1; 6; 7; falhei
↦(caminho-v2 1)
1; 6; 7; 11; 16; 21; 22; falhei
```