Linguagem OCaml

Algumas características

- O OCaml foi desenvolvido no INRIA, a partir de 1991, por Xavier Leroy e Damien Doligez, a que se juntaram Jérôme Vouillon, Didier Rémy e outros em 1996.
- A linguagem OCaml é uma das muitas linguagens que derivam da linguagem ML. [O ML começou por ser a meta-linguagem do demonstrador de teoremas LCF, desenvolvido no final dos anos 70 por Robin Milner e outros na University of Edinburgh. Mas o ML evoluiu para passar a ser usado como linguagem de programação"normal". Eis alguns dialetos da linguagem ML: Standard ML, Caml, OCaml, Alice, F#.]
- O OCaml é uma linguagem onde se unificam os paradigmas funcional, imperativo e orientado pelos objetos. Na nossa cadeira estamos interessados em estudar apenas a parte funcional pura da linguagem.
- Tem um sistema de tipos estático com suporte para polimorfismo e inferência de tipos.
- As implementações de OCaml colocam um grande ênfase na velocidade de execução. A velocidade é superior à do C++, para programas orientados pelos objetos.
- Tipos básicos: caracteres, inteiros, reais, booleanos.
- Tipos derivados: funções, listas, tuplos, registos, tipos soma.
- Tem gestão automática de memória.
- Modularidade e compilação separada.
- Biblioteca.
- A linguagem não tem nenhum padrão reconhecido por um organismo oficial normativo. A implementação do INRIA funciona como padrão *de facto*.

Elementos essenciais da linguagem OCaml

- O OCaml é uma linguagem multi-paradigma, mas nesta cadeira vamos usar apenas o sua componente funcional, para aprender o estilo de programação funcional, sem distrações.
- Um programa é uma sequência de funções. As funções podem ser recursivas.
- São dois os principais mecanismos que podem ser usados na escrita do corpo das funções em OCaml:
  - APLICAÇÃO (aplicação duma função a argumentos). Ex: `sqrt 9`
  - IF-THEN-ELSE. Ex: `if prime x then x else x/2`

Exemplo de função que usa os dois mecanismos:

```
let rec fact x =
    if x = 0 then 1 else x * fact (x-1)
;;
```

Avaliação de expressões em OCaml

A ideia de **avaliação** está no centro do processo de execução de programas funcionais. Merece pois a nossa melhor atenção.

Em OCaml as expressões são avaliadas usando uma estratégia chamada *call-by-value*: uma funções só é aplicada aos seus argumentos depois de eles terem sido avaliados (ou, mais simplesmente, uma função só pode ser aplicada a valores). Esta é a estratégia usada na maioria das linguagens incluindo: Java, C, C++, Pascal, etc.

Exemplo de avaliação:

```
(fun x -> x+1)(2+3)
= (fun x -> x+1)5
= 5+1
= 6
```

Exemplo de avaliação que não termina. Considere a função `loop`, assim definida:

```
let rec loop x = loop x ;;
```

Avaliação:

```
(fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= ...  não termina
```

Mais uma avaliação. Considere a função `fact`, assim definida:

```
let rec fact x =
    if x = 0 then 1 else x * fact (x-1)
;;
```

Avaliação:

```
fact 3 =
= 3 * fact 2
= 3 * (2 * fact 1)
= 3 * (2 * (1 * fact 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

Tipos em OCaml

Um **tipo** representa uma coleção de valores e tem associados um conjunto de **literais** e um conjunto de **operações**. (Um literal é uma expressão que não precisa de ser avaliada e denota um valor particular.)

Tipos básicos

TIPO	LITERAIS	OPERAÇÕES MAIS USADAS
int	5 -456	+ - * / mod min_int max_int int_of_float
float	3.14e-21	+. -. *. /. sqrt exp log sin ceil floor float_of_int
string	"" "ola"	^ String.length String.sub String.get
bool	false true	not || &&
char	'a' '$'	int_of_char char_of_int
unit	()	ignore

Tipos compostos

TIPO	LITERAIS	OPERAÇÕES MAIS USADAS
'a->'b	(fun x -> x+1)	aplicação
'a*'b	(5, 5.6)	fst snd *emparelhamento de padrões*
'a list	[] [3;5;7]	*emparelhamento de padrões*
tipos produto	*diversos*	*. emparelhamento de padrões*
tipos soma	*diversos*	*emparelhamento de padrões*

Inferência de tipos

O tipo dos argumentos e do resultado das funções não se declaram em OCaml. A implementação faz **inferência de tipos**: ela infere para cada função o tipo mais geral que é possível atribuir a essa função.

Qual o tipo das seguintes funções anónimas?

```
fun x -> x+1                    : int -> int
fun x -> x +. 1.0               : float -> float
```

```
fun x -> x ^ x                    : string -> string
fun (x,y) -> x + y                : (int * int) -> int
fun (x,y) -> (y,x)                : ('a*'b) -> ('b*'a)
fun x y -> (y,x)                  : 'a -> 'b -> ('b*'a)
```

As quatro primeiras funções dizem-se **monomórficas** porque só aceitam argumentos de tipos fixos.

A duas últimas funções dizem-se **polimórficas** pois aceitam argumentos de tipos diversos.

De todas estas funções, a última é a única que aceita mais do que um argumento, neste caso dois. A razão pela qual o seu tipo tem duas setas será estudada mais tarde.

Não há sobreposição (overloading)

A linguagem OCaml não suporta sobreposição (overloading) de nomes ou operadores. Por exemplo, em OCaml o operador "+" denota apenas a soma inteira (a soma real denota-se "+.").

Para contrastar, as linguagens C, C++ e Java suportam sobreposição. Por exemplo, em Java o operador "+" é usado para denotar três operações: soma de inteiros, soma de reais, concatenação de strings.

Operadores

Para alguns dos operadores mais usados em OCaml, eis uma tabela de correspondência entre o OCaml e o Java:

OCaml	**Java**
&&	&&
||	||
not	!
mod	%
=	==
<>	!=
+	+
+.	+
^	+
-	-
-.	-

Funções com múltiplos argumentos em OCaml

Em OCaml há duas formas de escrever funções com mais do que um argumento: a **forma não-curried** e a **forma curried**. A segunda forma é mais elaborada e a que tem mais vantagem técnicas.

Forma não-curried

Os vários argumento agrupam-se num único tuplo ordenado. Exemplos:

```
let nAdd (x,y) = x+y ;;
nAdd: (int * int) -> int

let nAdd3 (x,y,z) = x+y+z ;;
nAdd3: (int * int * int) -> int

nAdd (3,4) = 7
nAdd3 (2,8,1) = 11
```

Tecnicamente, a função `nAdd` tem apenas um parâmetro, de tipo `int*int`. Mas claro, através desse único parâmetro conseguimos passar duas unidades de informação.

Forma curried

Os argumentos ficam separados. Exemplos:

```
let cAdd x y = x+y ;;
cAdd: int -> int -> int

let cAdd x y z = x+y+z ;;
cAdd: int -> int -> int -> int

cAdd 3 4 = 7
```

```
cAdd 2 8 1 = 11
```

Escrever *funções curried* não tem nada que saber: basta usar espaços em branco a separar os parâmetros, na cabeça de cada função. Contudo o tipo destas funções afigura-se surpreendente para quem o observa pela primeira vez. Isso é resultado da representação interna das funções em OCaml. Segue-se a explicação...

Representação interna das funções em OCaml

Por uma questão de simplicidade e regularidade de implementação, o OCaml só usa internamente funções anónimas com um único argumento. Uma função com múltiplos argumentos é convertida para um formato interno especial - chamado **forma interna** - que envolve apenas funções anónimas com um único argumento.

Por exemplo, a função

```
let cAdd x y = x+y ;;
```

é internamente convertida em:

```
let cAdd = fun x -> (fun y -> x+y) ;;
```

Repare na engenhosa a ideia que está por detrás do esquema de tradução.

Vejamos mais alguns exemplos de tradução, lado a lado:

```
let cAdd x y z = x+y+z ;;      let cAdd = fun x -> (fun y -> (fun z -> x+y+z)) ;;

let f x = x+1 ;;               let f = fun x -> x+1 ;;

let nAdd (x,y) = x+y ;;        let nAdd = fun (x,y) -> x+y ;;
```

Avaliação de expressões envolvendo funções com múltiplos argumentos

Compare a avaliação das duas seguintes expressões:

```
nAdd (2,3)
= (fun (x,y) -> x+y) (2,3)
= 2 + 3
= 5

cAdd 2 3
= (fun x -> (fun y -> x+y)) 2 3
= (fun y -> 2+y) 3
= 2 + 3
= 5
```

Aplicação parcial

As funções curried têm a vantagem de poderem ser **aplicadas parcialmente** ou seja, poderem ser invocadas omitindo alguns dos argumentos do final. Eis um exemplo de aplicação parcial:

```
let succ = cAdd 1 ;;
succ: int -> int
```

Associatividades

- O **operador de aplicação** é associativo à esquerda. (Note que este operador é *invisível*, pois nunca se escreve).
  - Portanto, a expressão `f a b` deve ser interpretada como `(f a) b`.
- O **construtor de tipos funcionais ->** é associativo à direita.
  - Portanto, o tipo `int -> int -> int` deve ser interpretado como `int -> (int -> int)`.

Formas equivalentes de escrever a mesma função

As seguintes quatro declarações são equivalentes, no sentido em que declaram exatamente a **mesma** função `f:int->int->int`.

```
let f x y = x + y ;;                    (* formato externo preferido *)
let f x = (fun y -> x+y) ;;             (* formato externo *)
let f = (fun x y -> x+y) ;;             (* formato externo *)
let f = (fun x -> (fun y -> x+y)) ;;    (* formato interno *)
```

Todas são convertidas para a mesma forma interna.

Funções como valores de primeira classe

Nas linguagens funcionais as funções têm o estatuto de **valores de primeira classe**. Isso significa que as funções têm um estatuto tão importante como o dos inteiros, reais, e outros tipos predefinidos.

Concretamente, numa linguagem funcional as funções podem ...

- Ser passadas como argumento para outras funções;
- Podem ser retornadas por outras funções;
- Podem ser usadas como elementos constituintes de estruturas de dados;
- Têm uma representação literal própria. Exemplo: `(fun x->x+1)`

Exemplos

Exemplo 1. Composição de funções.

```
let compose f g = fun x -> f (g x) ;;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Também se pode escrever:

```
let compose f g x = f (g x) ;;
```

Exemplo 2. Função para converter do formato não-curried para o formato curried.

```
let curry f = fun x -> fun y -> f (x,y) ;;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Também se pode escrever:

```
let curry f x y = f (x,y) ;;
```

Exercício: Escrever a função inversa `uncurry`.

Exercício: Que funções são as seguintes e quais os seus tipos:

- `uncurry compose`
- `compose curry uncurry`
- `compose curry uncurry`

Exemplo 3. Como representar conjuntos usando apenas funções?

Um conjunto é uma entidade cuja principal característica é a possibilidade de se poder saber se um valor lhe pertence ou não lhe pertence. Assim vamos representar cada conjunto por uma função booleana que aplicada a um valor produz `true` se esse valor pertence ao conjunto e `false` se não pertence. Esta é a chamada **função característica** do conjunto.

- Conjunto vazio:

```
let set0 = fun x -> false ;;
```

- Conjunto universal:

```
let setu = fun x -> true ;;
```

- Criação de conjunto singular:

```
let set1 x = fun y -> y = x ;;
```

Exercício: Usando esta representação, escrever as funções `belongs`, `union` e `intersection`.

Exemplo 4. Estrutura de dados com funções: lista de funções.

```
let mylist = [(fun x -> x+1); (fun x -> x*x)] ;;
```

Uma limitação das funções

No caso geral, saber se duas funções dadas são iguais (i.e. saber se produzem sempre os mesmos resultados para os mesmos argumentos) é um problema que não pode ser resolvido por computador. Numa tal situação, costuma dizer-se que o problema é indecidível.

```
# (fun x -> x+1) = (fun x -> x+1) ;;
Exception: Invalid_argument "equal: functional value".
```

Listas homogéneas em OCaml

Apresentam-se aqui os três aspetos essenciais relativos as listas em OCaml: como se escrevem listas literais; como se constroem novas listas a partir de listas mais simples; como se analisam listas e se extraem os seus elementos constituintes.

Listas literais

Exemplos de listas literais:

```
[]                                  : 'a list
[2;4;8;5;0;9]                       : int list
["ola"; "ole"]                       : string list
[[1;2]; [4;5]]                      : int list list
[(fun x -> x+1); (fun x -> x*x)]    : (int->int) list
```

Construtores de listas

Estão disponíveis dois construtores de listas que, como o nome indica, servem para construir listas novas:

```
[] : 'a list
:: : 'a -> 'a list -> 'a list
```

- O construtor `[]` chama-se "lista vazia" e representa a lista vazia.
- O construtor `::` chama-se "cons" e serve para construir listas não vazias.

O operador :: é associativo à direita.

Exemplos de utilização de `cons`:

```
2::[3;4;5] = [2;3;4;5]
1::2::3::[] = [1;2;3]
[]::[] = [[]]
[1;2]::[3;4] = ERRO
4::5 = ERRO
[1;2]::[[3;4;5]] = [[1;2];[3;4;5]]
```

Processamento de listas usando emparelhamento de padrões

O processamento de listas efetuar-se por *análise de casos*, usando a construção **match** e *padrões*. Exemplo:

```
(* len : 'a list -> int *)

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

A função anterior trata dois casos, cada um dos quais tem um padrão diferente associado. Os vários casos são analisados sequencialmente, de cima para baixo.

Mais um exemplo. A seguinte função aplica-se a uma lista de pares ordenados e troca entre si as componentes de cada par:

```
(* swapPairs : ('a * 'b) list -> ('b * 'a) list *)

let rec swapPairs l =
    match l with
      [] -> []
    | (x,y)::xs -> (y,x)::swapPairs xs
;;
```

Método indutivo

Redução de problemas a problemas mais simples

Considere as seguintes duas funções recursivas, escritas em ML:

```
let rec fact n =
    if n = 0 then 1
    else n * fact (n-1)
;;

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Analisando estas duas funções recursivas verificamos que quando lidam com o caso não-trivial - o chamado **caso geral** - ambas efetuam **reduções de problemas a problemas mais simples**. Concretamente, a função `len` reduz o problema `len (x::xs)` ao problema mais simples `len xs`, e a função `fact` reduz o problema `fact n` ao problema mais simples `fact (n-1)`.

Além de efetuarem *redução de problemas*, as duas funções lidam também, separadamente, com o caso trivial de cada problema - o chamado **caso base**. Concretamente, a função `len` trata separadamente o caso base `len []` e a função `fact` trata separadamente o caso base `fact 0`.

Repare que quando uma destas duas funções é aplicada a um argumento, inicia-se uma sucessão de *reduções a problemas mais simples* que só termina quando o caso base é alcançado. Por exemplo, a sequência de *reduções* produzida pela aplicação `len [7;6;5;4]` é a seguinte:

```
len [7;6;5;4]              <-- problema inicial
= 1 + len [6;5;4]          <-- problema um pouco mais simples
= 1 + 1 + len [5;4]        <-- problema ainda mais simples
= 1 + 1 + 1 + len [4]      <-- problema ainda mais simples
= 1 + 1 + 1 + 1 + len []   <-- caso base
= 1 + 1 + 1 + 1 + 0
= 4
```

As funções recursivas `len` e `fact` são representativas do que é programar usando o núcleo funcional do ML. Em ML programa-se essencialmente *reduzindo problemas a problemas mais simples*.

Técnica do método indutivo

O método indutivo (também conhecido como técnica da divisão e conquista) serve para ajudar a programar funções recursivas que efetuam *reduções de problemas a problemas mais simples*.

O método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA (para começar a raciocinar) que o resultado S de aplicar a função a argumentos *mais simples do que os iniciais* (por exemplo a cauda da lista-argumento) já se encontra disponível.
- O problema que é então realmente preciso então resolver é o seguinte: COMO SE PASSA DE S PARA G?

Exemplo de utilização do método indutivo

Problema: Programar uma função `len` que determine o comprimento de listas.

Resolução: Para começar, a função `len` recebe uma lista, a qual terá de ser processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral `G=len (x::xs)`, vamos começar por assumir que já temos o problema resolvido para o caso, mais simples, `S=len xs`.

Desde já podemos ir escrevendo a seguinte estrutura incompleta:

```
let rec len l =
    match l with
        [] -> ...
      | x::xs -> ... len xs ...
;;
```

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de `G=len (x::xs)` a partir do valor de `S=len xs`?* Mas este problema é simples! De facto, basta adicionar uma unidade a S para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

NOTA1: Neste exemplo, reduzimos o problema `G=len (x::xs)` ao problema `S=len xs`. Mas esta não é a única redução possível... Existem muitas outras... Voltaremos a este assunto noutra aula.

NOTA2: A função `len` é tão simples que faz o método indutivo parecer óbvio e desinteressante. No entanto, este método tem imensas virtualidades:

- Ajuda-nos a organizar o pensamento quando queremos resolver problemas mais complicados, e.g. funções `power` e `sort`.
- Simplifica a resolução de problemas. [Repare que já não temos de resolver "o problema completo": passamos a ter de resolver "apenas" o problema da passagem de S para G, o que costuma ser "muito mais fácil".]
- Ajuda-nos a descobrir soluções simples e compactas.

Mais funções sobre listas programadas usando o método indutivo

Estude-as todas de forma muito atenta. Depois tente programá-las todas "sem espreitar".

Comprimento de lista:

```
len : 'a list -> int

let rec len l =
    match l with
```

```
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

Soma dos elementos de lista:

```
sum : int list -> int

let rec sum l =
    match l with
        [] -> 0
      | x::xs -> x + sum xs
;;
```

Concatenação de listas:

```
append : 'a list -> 'a list -> 'a list

let rec append l1 l2 =
    match l1 with
        [] -> l2
      | x::xs -> x::append xs l2
;;
```

Acrescentar elemento no final de lista:

```
putAtEnd : 'a -> 'a list -> 'a list

let rec putAtEnd v l =
    match l with
        [] -> [v]
      | x::xs -> x::putAtEnd v xs
;;
```

Inversão de lista:

```
rev : 'a list -> 'a list

let rec rev l =
    match l with
        [] -> []
      | x::xs -> append (rev xs) [x]
;;
```

Máximo duma lista (tratamento implícito do erro):

```
maxList : 'a list -> 'a

let rec maxList l = (* pre: l <> [] *)
    match l with
        [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Máximo duma lista (tratamento explícito do erro):

```
maxList : 'a list -> 'a

let rec maxList l =  (* pre: l <> [] *)
    match l with
        [] -> raise (Arg.Bad "maxList: lista vazia")
      | [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Aplicação de função a todos os elementos de lista:

```
map : ('a -> 'b) -> 'a list -> 'b list

let rec map f l =
    match l with
        [] -> []
      | x::xs -> (f x)::map f xs
;;
```

Seleção dos elementos duma lista que verificam uma dada propriedade:

```
filter : ('a -> bool) -> 'a list -> 'a list

let rec filter b l =
    match l with
        [] -> []
      | x::xs ->
            if b x then x::filter b xs
            else filter b xs
;;
```

Concatenação de todas as listas contidas numa lista de listas:

```
flatten : 'a list list -> 'a list

let rec flatten ll =
    match ll with
        [] -> []
      | l::ls ->
            l @ flatten  ls
;;
```

Concatenação de todos os resultados gerados por uma função de mapping que produz listas:

```
flatMap : ('a -> 'b list) -> 'a list -> 'b list

let rec flatMap f l =
    match l with
        [] -> []
      | x::xs ->
            (f x) @ flatMap f xs
;;
```

Ordenação de listas:

```
sortList : 'a list -> 'a list

let rec insOrd v l =
    match l with
        [] -> [v]
      | x::xs ->
          if v <= x then v::x::xs
          else x::insOrd v xs
;;

let rec sortList l =
    match l with
        [] -> []
      | x::xs ->
          insOrd x (sortList xs)
;;
```

Fusão de listas ordenadas sem repetições (aqui dá jeito usar matching duplo):

```
fusion : 'a list -> 'a list -> 'a list

let rec fusion l1 l2 =
    match l1, l2 with
        _, [] -> l1
      | [], _ -> l2
      | x::xs, y::ys ->
          if x < y then x::fusion xs l2
          else if y < x then y::fusion l1 ys
          else x::fusion xs ys
;;
```

Exemplos de avaliações

```
sum [1;2;3]
= 1 + sum [2;3]
= 1 + 2 + sum [3]
= 1 + 2 + 3 + sum []
= 1 + 2 + 3 + 0
= 6
```

```
append [1;2;0] [3;4]
= 1::append [2;0] [3;4]
= 1::2::append [0] [3;4]
= 1::2::0::append [] [3;4]
= 1::2::0::[3;4]
= [1;2;0;3;4]

rev [1;2;3]
= (rev [2;3]) @ [1]
= (rev [3]) @ [2] @ [1]
= (rev []) @ [3] @ [2] @ [1]
= [] @ [3] @ [2] @ [1]
= [3;2;1]
```

A função @

A função `append` está predefinida em ML, sendo denotada pelo operador binário `@`. Assim, podemos escrever:

```
[1;2;3] @ [4;5;6] = append [1;2;3] [4;5;6] = [1;2;3;4;5;6]
```

No entanto, a função `@` é pouco eficiente e deve ser usada com critério. Por exemplo, para acrescentar um zero à cabeça duma lista `list` é má ideia escrever `[0]@list`; escreve-se sempre `0::list`. No entanto, para acrescentar um zero ao final duma lista `list` não temos alternativa e escreve-se mesmo `list@[0]`.

Nomes locais

Construção let-in

A construção **let-in**:

```
let n = exp in
    exp1
```

associa o nome `n` ao valor da expressão `exp` no contexto da expressão `exp1`. O nome `n` passa a definir uma constante local à expressão `exp1`.

O nome `n` também pode ser parametrizado, assim

```
let n x = exp in
    exp1
```

mas repare que esta forma não introduz nada de realmente novo por ser equivalente a

```
let n = (fun x -> exp) in
    exp1
```

A palavra reservada **and** pode ser usada para definir simultaneamente vários nomes no mesmo `let-in`. Por exemplo:

```
let a = 1 and b = 2 in
    a + b ;;
```

ou

```
let f x = x + 1 and g x = x + 2 in
    f (g x) ;;
```

Exercício: Procure no manual da linguagem a forma sintática geral da construção **let-in**.

Vamos ver de seguida que a construção **let-in** permite:

1. Aumentar a legibilidade de certas função;
2. Aumentar a eficiência de certas função;
3. Definir funções mutuamente recursivas.

1. Legibilidade

Considere o seguinte problema: `Escreva em ML uma função chamada` **`parque`** `que calcule o preço a pagar no parque de estacionamento subterrâneo dos Restauradores, em Lisboa. A função parque recebe 2 pares ordenados de inteiros como argumentos: hora e minuto de entrada, hora e minuto de saída. Os tempos de permanência são arredondados para horas completas e sempre para cima. Assuma que o carro nunca está no parque mais do que 24:00. Em Março de 1999, a tabela de preços era a seguinte:`

```
1ª hora         120 cêntimos
2ª hora         140 cêntimos
3ª hora         150 cêntimos
seguintes       155 cêntimos
```

Apresentam-se a seguir duas versões equivalentes da função `parque`, a segunda programada usando **let-in**. A escolha entre elas é, em grande medida, uma questão de gosto, mas pode argumentar-se que a segunda versão, apesar de mais longa, é mais fácil de perceber porque torna explícita a ordem de avaliação das sub-expressões dentro duma expressão que é demasiado grande e complexa.

```
let parque (he, me) (hs, ms) =
    pagar (convHoras (permanencia (convMinutos he me) (convMinutos hs ms)))
;;
let parque (he, me) (hs, ms) =
    let te = convMinutos he me in
        let ts = convMinutos hs ms in
            let perm = permanencia te ts in
                let h = convHoras perm in
                    pagar h
;;
```

Exercício: As funções auxiliares não foram ainda programadas. Tente escrevê-las.

2. Eficiência

A construção **let-in** também pode ser usada para aumentar a eficiência de certas funções. Por exemplo a segunda função é mais eficiente do que a primeira (porquê?):

```
let f g x =
    g x + g x * g x
;;

let f g x =
    let r = g x in
        r + r * r
;;
```

3. Funções mutuamente recursivas

A forma geral da construção **let-in** inclui a possibilidade de utilização simultânea de **rec** e **and**. Isso permite definir 2 ou mais **funções mutuamente recursivas**. Eis um exemplo curioso com duas funções:

```
    let rec even n = if n = 0 then true else odd (n-1)
         and odd n = if n = 0 then false else even (n-1) in
             exp
;;
```

O que fazem estas funções?

Tradução do let-in

A construção **let-in**

```
let n = exp in
    exp1
```

é internamente traduzida para a seguinte representação:

```
(fun n -> exp1) exp
```
As duas formas são equivalentes (medite um pouco nisso!), mas a primeira é mais fácil de perceber.

Tradução do let simples

A já nossa conhecida construção **let-simples**, que permite definir nomes globais, é internamente traduzida para a construção **let-in** - os nomes globais são habilidosamente traduzidos para nomes locais.

Por exemplo, a seguinte sequência de definições e expressões:

```
# let f x = x + 1 ;;
# let g x = x + 2 ;;
# f (g x) ;;
```
é internamente traduzida para um conjunto de **let** aninhados:

```
let f x = x + 1 in
    let g x = x + 2 in
        f (g x) ;;
```

Tipos soma (uniões)

Muitas linguagens de programação incluem uma construção específica para a definição de tipos cujos valores podem assumir **diferentes variantes**, disjuntas entre si. Essa construção designa-se genericamente por **tipo soma**.

Por exemplo, numa linguagem com suporte para tipos soma é possível definir um tipo de dados *cshape*, para representar formas geométricas coloridas cujos valores podem assumir as três seguintes variantes: *Line*, *Circle* e *Rect*.

Um tipo soma permite conciliar o **geral** com o **particular**. Por um lado os elementos das variantes *Line*, *Circle* e *Rect* são formas geométricas coloridas com algumas propriedades comuns: no mínimo todas têm um cor! Por outro lado são variantes, cada uma delas com dados específicos associados: um círculo tem um centro e um raio, uma linha tem dois pontos extremos, etc.

Os tipos soma do Pascal chamam-se *registos com variante*.
Os tipos soma do C são as *uniões*.
Os tipos soma do Java, Smalltalk e C++ são as *classes abstratas* (imagine uma classe abstrata *cshape* com três subclasses concretas *Line*, *Circle* e *Rect*). [Em rigor, as classes abstratas são um bocadinho diferentes dos tipos soma: os tipos soma são entidades fechadas enquanto que as classes abstratas são extensíveis.]

Como é em ML?

Tipos soma em ML

Para exemplificar, o tipo soma cshape atrás referido pode ser definido em ML da seguinte forma, usando os tipos auxiliares color e point:

```
type color = int ;;
type point = float*float ;;

type cshape = Line of color*point*point
            | Circle of color*point*float
            | Rect of color*point*point ;;
```
Repare que o nome dos tipos definidos pelo utilizador começa sempre por letra minúscula e o nome de cada variante começa sempre por letra maiúscula. Chamam-se **etiquetas**, aos nomes das variantes.

Continuando a usar o mesmo exemplo, vejamos agora quais são os mecanismos essenciais para escrever e manipular valores de tipos soma.

Literais: Eis dois literais de tipo cshape. Repare como o nome de cada variante é usado para marcar os respetivos literais.

```
let a = Line (34658, (2.5, 7.8), (-24.005, 1000.0001)) ;;
let b = Circle (11111, (-24.005, 1000.0003), 3.1233333) ;;
```

Construção: Por exemplo, a seguinte função cria círculos centrados no ponto zero:

```
let zeroCircle c r = Circle (c, (0.0, 0.0), r) ;;
```

Processamento: Eis uma função que calcula a área duma forma colorida. Os elementos de qualquer tipo soma são processados usando emparelhamento de padrões.

```
let area cs =
    match cs with
        Line   (_, _, _) -> 0.0
      | Circle (_, _, r) -> 3.14159 *. r *. r
      | Rect   (_, (tx,ty), (bx,by)) -> abs_float ((bx -. tx) *. (by -. ty))
;;
```

Eis uma função que determina o raio duma forma. Só está definida para círculos.

```
let radius (Circle (_, _, r)) = r ;;
```

Alguns tipos soma predefinidos em ML

1. O tipo **'a list** é um tipo soma. Internamente a sua definição assemelha-se a:
   ```
   2. type 'a list = Nil | Node of 'a * 'a list ;;
   ```
3. O tipo **bool** também é um tipo soma, internamente é definido da seguinte forma:
   ```
   4. type bool = false | true ;;
   ```

   O ML abre uma exceção neste caso, e permite que o nome das variantes comece por letra minúscula.

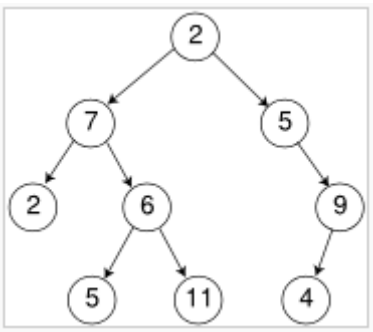
5. O tipo **'a option** é um tipo soma que permite representar o conceito de **ausência de valor**, em situações que tal possa ser útil. Internamente, é definido assim:
   ```
   6. type 'a option = None | Some of 'a ;;
   ```

Os três papéis das etiquetas dum tipo soma

As etiquetas dum tipo soma tem três papéis:

- Na definição do tipo, Identificam os vários ramos.
- Denotam os *construtores* que o sistema cria automaticamente para o tipo em causa. Um *construtor* é uma função especial que gera valores dum tipo soma. Quando escrevemos Circle(111, (0.0, 0.0), 12.4) estamos a chamar um construtor das nossas formas.
- São elementos constituintes de novos padrões que a linguagem passa a reconhecer automaticamente.

Árvores binárias



Em programação, as árvores binárias permitem exprimir informação hierarquizada e permitem organizar dados por forma a aumentar a velocidade de acesso a eles.

O tipo soma **árvore binária** não está predefinido em ML, mas é fácil de definir usando um tipo soma:

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;
```

Literais: Eis uma constante do tipo `int tree`:

```
Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)) ;;
```

Construção: Por exemplo, a seguinte função cria folhas da árvore:

```
let makeLeaf v = Node(v, Nil, Nil) ;;
```

Processamento: Eis uma função que determina o número total de nós duma árvore binária:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
            1 + size l + size r
;;
```

Eis um exemplo de avaliação duma expressão envolvendo uma chamada da função "size":

```
  size (Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)))
= 1 + size (Node(2,Nil,Nil)) + size (Node(3,Nil,Nil))
= 1 + (1 + size Nil + size Nil) + (1 + size Nil + size Nil)
= 1 + (1 + 0 + 0) + (1 + 0 + 0)
= 1 + 1 + 1
= 3
```

Vocabulário relativo a árvores

- Raiz - nó sem ascendentes
- Nó interno - nó diferente da raiz com pelo menos um filho
- Folha - nó sem filhos

Método indutivo aplicado a árvores binárias

No caso das árvores binárias, o método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR que os resultados L, R de aplicar a própria função a argumentos *mais simples do que os iniciais* (por exemplo às duas sub-árvores) já se encontram disponíveis. O problema que é então preciso resolver é o seguinte: COMO É QUE A PARTIR DE L E R SE CHEGA A G?

Exemplo de utilização do método indutivo

Problema: Programar uma função "size" que determine o número total de nós duma árvore binária:

Resolução: A função "size" aplica-se a uma árvore, a qual será, naturalmente, processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral "G=Node(x,l,r)", vamos começar por assumir que já temos o problema resolvido para os casos "L=size l" e "R=size r". Obtemos assim o seguinte PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR:

```
 let rec size t =
    match t with
         Nil ->  ...
       | Node(x,l,r) ->
            ... size l ... size r ...
;;
```

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de "G=size (Node(x,l,r))" a partir dos valores "L=size l" e "R=size r"?* Mas este problema é simples. De facto, basta adicionar uma unidade a L+R para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
            1 + (size l) + (size r)
;;
```

Mais duas funções sobre árvores binárias

Altura duma árvore:

```
(* height: 'a tree -> int *)

let rec height t =
    match t with
       Nil -> 0
     | Node(x,l,r) ->
           1 + max (height l) (height r)
;;
```

Árvore ao espelho:

```
(* mirror: 'a tree -> 'a tree *)

let rec mirror t =
    match t with
      Nil -> Nil
    | Node (x,l,r) ->
           Node (x,mirror r,mirror l) (* o r e l trocam de posição *)
;;
```

Árvores n-árias

Numa árvore n-ária, cada nó pode ter um número qualquer (ilimitado) de filhos.

O tipo soma **árvore n-ária** pode definir-se assim em ML:

```
type 'a ntree = NNil | NNode of 'a * 'a ntree list ;;
```

Literais: Eis uma constante de tipo "int ntree":

```
NNode(1, [NNode(2,[]); NNode(3,[]); NNode(4,[])]) ;;
```

Construção: Por exemplo, a seguinte função cria folhas da árvore:

```
let makeLeaf v = NNode(v, []) ;;
```

Processamento: A função `size` determina o número de nós duma árvore n-ária. Note que se usa uma função auxiliar que processa uma lista de árvores.

```
(* size: 'a ntree -> int *)

let rec lsize ts =
    match ts with
        [] -> 0
      | NNil::ts -> lsize ts
      | NNode(x,cs)::ts -> 1 + lsize cs + lsize ts
;;

let size t =
    lsize [t]
;;
```

No caso geral da função anterior, repare que há duas chamadas recursivas: uma para os filhos dum nó, e outra para os irmãos à direita desse nó.

Esquema geral da utilização do método indutivo no tratamento de árvores n-árias:

```
let rec lf ts =
    match ts with
        [] -> ...
      | NNil::ts -> ... lf ts
      | NNode(x,cs)::ts -> ... lf cs ... lf ts ...
;;
```

```
let f t =
    lf [t]
;;
```

Árvore ao espelho:

```
(* mirror: 'a ntree -> 'a ntree *)

let rec lmirror ts =
    match ts with
        [] -> []
      | NNil::ts -> lmirror ts @ [NNil]
      | NNode(x,cs)::ts -> lmirror ts @ [NNode(x,lmirror cs)]
;;

let mirror t =
    List.hd (lmirror [t])
;;
```

Padrões

Um **padrão** é uma expressão especial que representa um **conjunto de valores**.

A utilização de padrões torna as funções mais fáceis de escrever e de entender. Os padrões são, portanto, bons amigos do/a programador/a. As linguagens funcionais antigas (e.g. Lisp) não usavam padrões, mas as linguagens funcionais modernas (e.g. ML, Haskell) não os dispensam.

Exemplos de padrões:

```
Padrão          Conjunto de valores representados
~~~~~~          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[]              lista vazia
[x]             listas com um elemento
[x;y]           listas com dois elementos
x::xs           listas não vazias
x::y::xs        listas com pelo menos dois elementos
5::xs           listas cujo primeiro elemento é 5
x               todos os valores (padrão universal)
_               padrão universal anónimo
(x,y)           todos os pares ordenados
(0,y)           todos os pares ordenados cuja 1ª componente é 0
8               inteiro 8
(x,y)::xs       lista não vazia de pares ordenados
'a'..'z'        letras de 'a' a 'z'
```

Numa expressão **match** podem ocorrer padrões em número variável (ver exemplos 1 e 2). Durante a avaliação, esses padrões são explorados sequencialmente, de cima para baixo.

Exemplos

Nos exemplos que se seguem, os padrões aparecem assinalados a negrito.

Exemplo1:

```
let rec len l =
    match l with
        [] -> 0
      | _::xs -> 1 + len xs
;;
```

Exemplo2:

```
let rec count5 l =
    match l with
        [] -> 0
      | 5::xs -> 1 + count5 xs
      | _::xs -> count5 xs
;;
```

Exemplo 3:

```
fun (x,y) -> (y,x)
```

Exemplo 4:

```
let f (x,y) =
    (y,x)
;;
```

Exemplo 5:

```
let f g =
    let (x,y) = g 0 in
        x + y
```

Regra da seta ->

- No contexto que antecede a seta -> (portanto nas expressões **match** e nos argumentos das funções anónimas), as expressões são interpretadas como padrões. Por exemplo, nesse contexto, x::xs representa o conjunto de todas as listas não vazias.
- Fora do contexto que antecede a seta ->, as expressões são interpretadas da forma habitual. Por exemplo, fora desse contexto, x::xs representa a aplicação do construtor "cons" aos nomes x e xs).

Note que devido à forma como a construção **let-in** se define por tradução, toda a expressão que aparece imediatamente a seguir ao `let` também é interpretada como padrão. Também devido à forma como as funções com nome são traduzidas para para funções anónimas, toda a expressão que aparece no cabeçalho imediatamente antes do sinal de igual é interpretada como padrão.

Padrões disjuntos

Dois padrões dizem-se **disjuntos** se os conjuntos que eles representam são disjuntos. Numa expressão `match` a ordem dos casos só é irrelevante se os padrões forem disjuntos entre si.

Exemplos:

- Os dois padrões que ocorrem na função `len` **são disjuntos**:
  - `let rec len l =`
  - `match l with`
  - `[] -> 0`
  - `| x::xs -> 1 + len xs`
  - `;;`
- Os dois padrões que ocorrem na função `fact` **não são disjuntos**:
  - `let rec fact n =`
  - `match n with`
  - `0 -> 1`
  - `| n -> n * fact (n-1)`
  - `;;`

Emparelhamento

Como resultado do **emparelhamento dum valor com um padrão**, há duas consequências possíveis:

1. Falhanço;

2. Sucesso, e os nomes que ocorrem no padrão são ligados a valores.

Restrições

- Num padrão, não se permite a repetição de nomes. Por exemplo o padrão `(x,x)::xs` é inválido.
- Todos os nomes que ocorrem num padrão são considerados nomes novos, mesmo que esses nomes ocorram no ambiente envolvente.

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares com as duas componentes iguais:

```
let rec eqPairs l =
    match l with
      [] -> 0
    | (x,y)::xs ->    (* o padrão (x,x)::xs parece melhor, mas seria inválido *)
         if x=y then 1 + eqPairs xs
         else eqPairs xs
;;
```

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares em que a segunda componente é igual a um valor dado:

```
let rec countPairs v l =
    match l with
      [] -> 0
    | (x,y)::xs ->    (* o padrão (x,v)::xs parece melhor, mas não interessa pois contém
um v "novo" *)
         if y=v then 1 + countPairs v xs
         else countPairs v xs
;;
```

Tipos produto (tuplos e registos)

A maioria das linguagens de programação incluem nos seus sistemas de tipos uma construção específica que permite representar agrupamentos de dados heterogéneos. O nome genérico dessa construção, independente da linguagem particular, é **tipo produto**.

Numa linguagem com tipos produto é possível definir, por exemplo, um tipo de dados *pessoa* constituído por um nome (de tipo string), um ano de nascimento (de tipo int), uma morada (de tipo string), etc.

Os tipos produto do Pascal são os *registos*.
Os tipos produto do C são as *estruturas*.
Os tipos produto do Java, Smalltalk e C++ são as *classes*.
No Fortran, os tipos produto apareceram na versão Fortran 90 com o nome de *tipos derivados*.

Como é em ML?

Tipos produto em ML

Em ML há duas variedades de tipos produto: **tipos produto não etiquetados** e **tipos produto etiquetados**.

Tipos produto não etiquetados (tuplos)

Os produtos cartesianos do ML são exemplos de tipos produto, neste caso ditos **não-etiquetados**, e também conhecidos por **tuplos**. Por exemplo, para representar *pessoas*, pode usar-se em ML o seguinte tipos produto não etiquetado:

```
string * int * string
```

Literais: Para exemplificar, eis um valor do tipo anterior:

```
("João ", 1970, "Lisboa")
```

Construção: Para exemplificar vejamos uma função que muda a morada duma pessoa, criando um tuplo novo:

```
let moveTo (x,y,_) city = (x, y, city) ;;
```

Processamento: Como se processam tuplos? De duas formas:

- Usando emparelhamento de padrões:
  - `let getName p =`
  - `    match p with`
  - `      (x, _, _) -> x`
  - `;;`
- Ou usando as operações de acesso **fst** e **snd** predefinidas, se o registo tiver duas componentes:
  - `não aplicável ao nosso exemplo`

Tipos produto etiquetados (registos)

Mas o ML, também suporta **tipos produto etiquetados**, também conhecidos como **registos**, os quais requerem definição explícita. Eis um exemplo de tipo produto etiquetado:

```
type pessoa = { nome:string ; anoNasc:int ; morada:string } ;;
```

Literais: Eis um literal deste tipo:

```
{ nome = "João" ; anoNasc = 1970 ; morada = "Lisboa" }
```

Construção: Para exemplificar vejamos uma função que muda a morada duma pessoa, criando um registo novo:

```
let moveTo p city =
    { nome = p.nome ; anoNasc = p.anoNasc ; morada = city }
;;
```

Processamento: Como se processam registos? De duas formas:

- Usando emparelhamento de padrões:
  - `let getNome p =`
  - `    match p with`
  - `      { nome = x ; anoNasc = _ ; morada = _ } -> x`
  - `;;`
- Ou usando a operação de acesso "." e que funciona em ML exatamente como em Pascal ou C:
  - `let getNome p = p.nome ;;`

Tratamento de exceções em ML

Durante a execução de um programa, por vezes verificam-se determinadas condições (geralmente anómalas, mas nem sempre) às quais é necessário reagir alterando o fluxo de execução normal. Tais condições chamam-se **exceções**.

As exceções podem ser geradas:

- Ao nível do hardware. Por exemplo, em virtude duma divisão por zero.
- Ao nível do sistema operativo. Por exemplo, devido à impossibilidade de abrir um ficheiro inexistente, ou devido à tentativa de continuar a ler dum ficheiro que já chegou ao fim.
- Deliberadamente pelos próprios programas, usando a construção **raise**. Por exemplo, para lidar explicitamente com **argumentos proibidos**:
  - `let rec fact x =`
  - `    if x = 0 then 1`
  - `    else if x>0 then x * fact(x-1)`
  - `    else raise (Arg.Bad "fact")`
  - `;;`

Captura e tratamento de exceções

Quando uma exceção é gerada, o controlo da execução do programa é transferido para o **tratador de exceções** (*exception handler*) mais recentemente cativado. É abortada a avaliação de todas as funções chamadas depois da ativação desse tratador de exceções.

Em ML, um tratador de exceções escreve-se usando uma expressão **try-with**, como se exemplifica de seguida. A expressão `exp`, no seu interior, diz-se uma **expressão protegida**.

```
try
    exp
with Sys_error _ -> exp1
   | Division_by_zero -> exp2
   | End_of_file -> exp3
   ...
;;
```

Como é avaliada uma expressão `try-with`?

- Começa-se por avaliar a expressão protegida `exp`.
- Caso nenhuma exceção seja gerada por `exp`, então o `try-with` não tem qualquer efeito e a expressão global `try-with` tem o mesmo valor da expressão `exp`.
- Mas se for gerada alguma exceção por `exp`, então o `try-with` recebe o controlo da execução e usa emparelhamento de padrões para descobrir qual das expressões `exp1`, `exp2`, `exp3`, etc., deve ser avaliada em substituição de `exp`.
- Finalmente, se o emparelhamento de padrões anterior falhar, então a exceção é propagada para o `try-with` cronologicamente anterior.

Existe um tratador de exceções de sistema que apanha as exceções não tratadas e aborta a execução do programa com uma mensagem de erro apropriada a cada caso. Exemplos:

```
# 4/0;;
Exception: Division_by_zero.

# open_in "fdsg" ;;
Exception: Sys_error "fdsg: No such file or directory".
```

Definição de novas exceções

A lista de exceções predefinidas na linguagem encontra-se aqui: Index of exceptions.

O programador pode definir novas exceções. A sintaxe da definição duma nova exceção é igual à sintaxe da definição duma variante dum tipos soma.

Exemplos. O primeiro exemplo define uma exceção com argumento; o segundo define uma exceção sem argumento.

```
exception Stack_overflow of string * int ;;
exception I_m_so_out_of_here ;;
```

Funções parciais

Uma **função parcial** é uma função que só está definida em parte do seu domínio. (Não confundir com *aplicação parcial*, que é outra coisa.)

Por exemplo, a função `fact` é parcial porque só está definida para valores não-negativos:

```
let rec fact n = (* pre: n >= 0 *)
    if n = 0 then 1
    else n * fact (n-1)
;;
```

Outro exemplo: A função `maxList` é parcial porque só está definida para listas não-vazias:

```
let rec maxList l = (* pre: l <> [] *)
    match l with
        [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Quando se escreve uma função parcial, espera-se que essa função seja sempre aplicada a argumentos válidos. Mas os programas podem ter erros e é importante que os programas não disfarcem esses erros - é muito melhor a execução dum programar abortar do que terminar produzindo resultados errados. Por isso, convém garantir que a função não produz qualquer resultado quando aplicada a argumentos inválidos (por outras palavras, temos de obrigar o resultado a ser realmente *indefinido*).

A melhor forma de impedir uma função de produzir resultado, ao mesmo tempo gerando uma mensagem de erro clara, é gerar uma exceção. Assim:

```
let rec fact n =  (* pre: n >= 0 *)
    if n = 0 then 1
    else if n > 0 then n * fact (n-1)
    else raise (Arg.Bad "fact: numero negativo")
;;

let rec maxList l =  (* pre: l <> [] *)
    match l with
         [] -> raise (Arg.Bad "maxList: lista vazia")
        | [x] -> x
        | x::xs -> max x (maxList xs)
;;
```

Mas, repare, mesmo sem lançar exceções explícitas, as versões originais destas duas funções já garantem a não produção de resultados: a primeira aborta com "Stack overflow" e a segunda gera a exceção `Match_failure`.

Efeitos laterais

O ML usa um modelo de input/output imperativo, no qual as operações de input/output são tratadas como *efeitos laterais*.

O que é um **efeito lateral**? Um *efeito lateral* é qualquer atividade que uma função desenvolva para além de calcular um resultado a partir dos argumentos. Por exemplo:

- Escrever num ficheiro.
- Escrever no ecrã do computador.
- Ler dum ficheiro.

Como sabemos, no paradigma funcional é suposto uma função limitar-se a calcular um resultado a partir dos seus argumentos. O facto é que a linguagem ML ultrapassa os limites estritos do paradigma funcional, já que admite efeitos laterais nas funções.

Operador de sequenciação

Para sequenciar os efeitos laterais, o ML dispõe dum *operador de sequenciação* que se escreve ";" (como em Pascal!). Assim, por exemplo, para escrever no ecrã a string "ola", e logo a seguir a string "ole" usamos a seguinte função:

```
let hello () =
    print_string "ola" ; print_string "ole"
;;
```

Tecnicamente, ";" é o nome duma função com o seguinte tipo:

```
";": unit -> 'b -> 'b
```

e a seguinte definição interna:

```
";" x y = y ;;
```

Tipo unit e valor ()

Por vezes gostaríamos de definir funções sem argumentos ou sem resultados. Isso faz sentido para funções só com efeitos laterais. Mas o ML não o permite, pois obriga todas as funções a ter argumentos e um resultado. O tipo **unit** foi inventado para ser usado nessas situações.

O tipo **unit** é um tipo básico com apenas um valor, que se escreve **()**. Para comparação, note que o tipo **bool** é um tipo com apenas dois valores, que se escrevem **true** e **false**. Por seu lado, o tipo **void** do C é um tipo sem valores e por isso não resolveria o nosso problema em ML.

Exemplos de utilização de unit e ().

```
print_string: string -> unit
read_int: unit -> int
print_newline: unit -> unit

let x = read_int () in
    print_int (x+1)
```

Exemplo de função para escrever uma lista de inteiros no ecrã:

```
let rec printList l =
    match l with
       [] -> ()
     | x::xs -> print_int x ; print_newline () ; printList xs
;;
```

Input-Output em ML

Canais

As operações sobre ficheiros são efetuadas através de **canais**. Os canais são valores dos tipos predefinidos *in_channel* e *out_channel*.

Os "canais" do ML correspondem às "streams" do Java.

Em ML existem três canais predefinidos que são automaticamente abertos quando o programa começa a correr:

- `stdin : in_channel`
- `stdout : out_channel`
- `stderr : out_channel`

Primitivas de input/output

- **Primitivas para escrita em stdout**
  - `print_char: char -> unit`
  - `print_string: string -> unit`
  - `print_int: int -> unit`
  - `print_float: float -> unit`
  - `print_newline: unit -> unit`
- **Primitivas para leitura de stdin**
  - `read_line: unit -> string`
  - `read_int: unit -> int`
  - `read_float: unit -> float`
- **Primitivas para abertura e fecho de canais**
  - `open_in: string -> in_channel`
  - `open_out: string -> out_channel`

    - `close_in: in_channel -> unit`
    - `close_out: out_channel -> unit`
- **Primitivas para escrita em canais**
    - `output_char: out_channel -> char -> unit`
    - `output_string: out_channel -> string -> unit`
    - `flush: out_channel -> unit`
- **Primitivas para leitura de canais**
    - `input_char: in_channel -> char`
    - `input_line: in_channel -> string`

Esta lista de primitivas não é exaustiva. Há mais primitivas listadas no manual de referência (ver Basic Operations).

Exemplo de função sobre ficheiros programada usando o método indutivo

Cópia de ficheiros:

```
(* copyChannel: copia o canal de input ci para o canal de output co *)

let rec copyChannel ci co =
    try
        let s = input_line ci in
            output_string co s ;
            output_string co "\n" ;
            copyChannel ci co
    with End_of_file -> ()
;;

(* copyFile: abre os ficheiros ni e depois usa a função copyChannel *)

let copyFile ni no =
    let ci = open_in ni in
        let co = open_out no in
            copyChannel ci co ;
            close_in ci ;
            close_out co
;;
```

Esquema geral de utilização do método indutivo no tratamento de ficheiros linha a linha:

```
let rec f ci =
    try
        let s = input_line ci in
            ... f ci ...
    with End_of_file -> ...
;;
```

Introdução aos módulos em ML

Nenhuma linguagem moderna dispensa um sistema de módulos. Um sistema de módulos permite:

- Agrupar definições relacionadas: por exemplo permite agrupar um tipo de dados com as suas operações associadas.
- Forçar uma forma coerente de nomear essas definições para evitar conflitos de nomes.
- Ocultar a representação interna dos dados e ocultar as operações auxiliares.
- Está na base duma conceção modular de software.

O sistema de módulos do OCaml é muito rico. Para não dedicarmos excessivo tempo a este assunto, vamos ignorar quase todos os aspetos técnicos e vocabulário especializado associados ao sistema. Focamos a nossa atenção no estudo de alguns cenários de utilização que, afinal, cobrem a maioria das necessidades práticas.

Utilização de módulos existentes

Dentro do interpretador `ocaml` estão sempre disponíveis todos os módulos de biblioteca do ML (Index of modules).

Mas para aceder a módulos criados pelo utilizador dentro do interpretador `ocaml`, usa-se a diretiva `#load` assim:

```
# #load "MySet.cmo" ;;
```

Para referenciar elementos dum módulo, podem usar-se referências qualificadas, como se exemplifica:

```
# Sys.os_type ;;
- : string = "Unix"

# Sys.command "ls -l" ;;
total 24
-rw-r--r-- 1 amd amd  259 2008-03-11 10:26 main.ml
-rw-r--r-- 1 amd amd  440 2008-03-11 10:28 MySet.cmi
-rw-r--r-- 1 amd amd  654 2008-03-11 10:28 MySet.cmo
-rw-r--r-- 1 amd amd  394 2008-03-11 10:11 MySet.ml
-rw-r--r-- 1 amd amd  164 2008-03-11 10:11 MySet.mli
- : int = 0

# List.rev [1;2;3] ;;
- : int list = [3; 2; 1]

#
```

Mas é bastante mais prático abrir (importar) primeiro o módulo, para depois não ter de usar prefixos qualificadores:

```
# open Sys ;;

# os_type ;;
- : string = "Unix"

# command "ls -l" ;;
total 24
-rw-r--r-- 1 amd amd  259 2008-03-11 10:26 main.ml
-rw-r--r-- 1 amd amd  440 2008-03-11 10:28 MySet.cmi
-rw-r--r-- 1 amd amd  654 2008-03-11 10:28 MySet.cmo
-rw-r--r-- 1 amd amd  394 2008-03-11 10:11 MySet.ml
-rw-r--r-- 1 amd amd  164 2008-03-11 10:11 MySet.mli
- : int = 0
#
```

Criação dum módulo aberto

Para criar um módulo aberto, no qual todas as definições são públicas, basta criar um ficheiro com o nome pretendido e extensão "ml", digamos "MySet.ml". Depois é necessário compilar o módulo usando o comando

```
ocamlc -c MySet.ml
```

sendo gerados os ficheiros "MySet.cmi" e "MySet.cmo".

Exemplo de utilização do módulo aberto MySet:

```
$ ocaml
        Objective Caml version 3.09.2

# #load "MySet.cmo" ;;
# open MySet ;;
# empty ;;
- : 'a list = []
# make [1;2;3] ;;
- : int list = [1; 2; 3]
#
```

Eis o conteúdo do ficheiro "MySet.ml":

```
(* Module body MySet *)

type 'a set = 'a list ;;

let empty = [] ;;

let rec belongs v l =
```

```
    match l with
      [] -> false
    | x::xs -> x = v || belongs v xs
;;

let rec union l1 l2 =
    match l1 with
      [] -> l2
    | x::xs -> (if belongs x l2 then [] else [x])@union xs l2
;;

let rec clear l =
    match l with
      [] -> []
    | x::xs -> let cl = clear xs in
                (if belongs x cl then [] else [x])@cl
;;

let make l =
    clear l
;;
```

Criação dum módulo fechado

Para ocultar a representação interna dos dados e as operações auxiliares é necessário criar adicionalmente um **ficheiro interface** "MySet.mli" onde se declaram todas as entidades públicas da forma que se pretende que sejam vistas do exterior. Depois compila-se o módulo assim

```
ocamlc -c MySet.mli MySet.ml
```

sendo gerados os ficheiros "MySet.cmi" e "MySet.cmo".

Exemplo de utilização do módulo fechado MySet:

```
$ ocaml
        Objective Caml version 3.09.2

# #load "MySet.cmo" ;;
# open MySet ;;
# empty ;;
- : 'a MySet.set = <abstr>
# make [1;2;3] ;;
- : int MySet.set = <abstr>
# clear [1;2;3] ;;
Unbound value clear
#
```

Eis o conteúdo do ficheiro "MySet.mli":

```
(* Module interface MySet *)

type 'a set                (* abstract *)
val empty : 'a set
val belongs : 'a -> 'a set -> bool
val union : 'a set -> 'a set -> 'a set
val make : 'a list -> 'a set
```

Repare que neste ficheiro interface omitimos a representação interna do tipo `'a set` assim como a declaração da função auxiliar `clear`. Repare ainda na subtileza do tipo da função `make` (recebe uma lista mas retorna um set).

Quando se esconde a representação interna dum tipo de dados, diz-se que esse tipo é **abstracto** ou **opaco**.

Funções recursivas

Uma função recursiva bem definida faz sempre uma *análise de casos* sobre os seus parâmetros. Por exemplo, a função recursiva "fib" considera três casos relativamente ao seu parâmetro "n":

```
let rec fib n =
     if n = 0 then 1
     else if n = 1 then 1
     else fib (n-1) + fib (n-2)
;;
```

Existem sempre dois géneros de casos que uma função recursiva tem sempre de tratar: *casos base* e *casos gerais*. Os **casos base** são aqueles que não conduzem, nem directa nem indirectamente, a chamadas recursivas da função; os **casos gerais** são aqueles que conduzem, directa ou indirectamente, a chamadas recursivas da função. A função "fib" trata dois casos base, n=0 e n=1, e trata um caso geral, n>=2.

Toda a função recursiva deve tratar pelo menos um caso base e um caso geral. Além disso todas as chamadas recursivas que estão associadas aos casos gerais devem corresponder **problemas mais simples** do que o problema que a função, no seu todo, resolve (*problema mais simples* significa *problema mais próximo de algum dos casos base*). Apenas este conjunto de condições garante terminação.

Usando o método indutivo

O **método indutivo**, introduzido na aula teórica 6, ajuda a programar os casos gerais das funções recursivas. Concretamente, dá alguma orientação sobre como fazer a *redução de problemas a problemas mais simples*.

Quando se usa o método indutivo, **devem-se ter em mente** apenas as propriedades lógicas do problema a resolver. **Não se devem ter em mente** quaisquer preocupações relacionadas com as propriedades operacionais da função que está a ser programada. Misturar do uso do método indutivo com preocupações de natureza operacional é uma receita para o desastre.

De qualquer forma, depois de programada a função, já não há problema em tentar compreendê-la operacionalmente. Pode mesmo valer a pena escrever uma avaliação da função para se ganhar confiança nela. Exemplo: avaliação de (len [1;5;3;8]):

```
len [1;5;3;8] =
    1 + len [5;3;8] =
    1 + 1 + len [3;8] =
    1 + 1 + 1 + len [8] =
    1 + 1 + 1 + 1 + len [] =
    1 + 1 + 1 + 1 + 0 =
    4
```

Categorias de funções recursivas

Vamos agora estudar 4 categorias de funções recursivas, todas programadas usando o método indutivo: funções de tipo 1, de tipo 2, de tipo 3 e de tipo 4.

Na avaliação da qualidade duma função, um dos critérios que surge em primeiro lugar é o da **facilidade em compreender a função**. As funções de tipo 1 e 2 tendem a ser as mais simples de perceber, mas nem todos os problemas podem ser resolvidos directamente usando apenas funções de tipo 1 e 2.

Funções de tipo 1

As **funções de tipo 1** são programadas usando o método indutivo e caracterizam-se pela seguinte propriedade:

- A estrutura da função baseia-se num dos esquemas rígidos predefinidos, indicados a seguir. Repare que em todos esses esquemas, os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Nesses esquemas, o PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR aparece sublinhado.

Orientação prática: Quando se tenta resolver um problema, convém começar por procurar construir uma função de tipo 1 já que estas são as mais fáceis de escrever e compreender, na maioria dos casos.

Inteiros

```
let rec f n =
    if n = 0 then ...
    else ... f (n-1) ...
;;
```

Listas

```
let rec f l =
    match l with
        [] -> ...
      | x::xs -> ... f xs ...
;;
```

Árvores binárias

```
let rec f t =
    match t with
        Nil -> ...
      | Node(x,l,r) -> ... f l ... f r ...
;;
```

Árvores n-árias

```
  let rec lf ts =
      match ts with
          [] -> ...
        | NNil::ts -> ... lf ts
        | NNode(x,cs)::ts -> ... lf cs ... lf ts ...
  ;;

  let f t =
      lf [t]
  ;;

  let f t =
      let r = lf [t] in
          if r = [] then NNil
          else List.hd r
  ;;
```

Ficheiros: leitura de sequência

```
let rec f ci =
   try
       let s = input_line ci in
           ... f ci ...
   with End_of_file -> ...
;;
```

Strings

```
let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
let rec f s =
    if s = "" then ...
    else
        let (x,xs) = cut s in
            ... f xs ...
;;
```

Exemplos de funções de tipo 1: fact, len, append, rev, belongs, union, power, height, size, zeros, treeToList, balanced, subTrees, etc. (em suma, a maioria das funções estudadas até ao momento).

Mais exemplos de funções de tipo 1:

```
stringAsList: converte string em lista de caracteres
    let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
    let rec stringAsList s =
        if s = "" then []
        else
            let (x,xs) = cut s in
                x::stringAsList xs
    ;;

sortList: ordena lista
    let rec insOrd v l =
        match l with
            [] -> [v]
          | x::xs ->
              if v <= x then v::x::xs
              else x::insOrd v xs
    ;;
    let rec sortList l =
        match l with
            [] -> []
          | x::xs ->
              insOrd x (sortList xs)
    ;;
```

Funções de tipo 2

As **funções de tipo 2** são programadas usando o método indutivo e caracterizam-se pelas duas seguintes propriedades:

- Os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.
- Fogem aos esquemas rígidos das funções de tipo 1.

Orientação prática: Quando se tenta escrever uma função de tipo 1, por vezes descobre-se que é necessário ou conveniente fazer alguns pequenos ajustamentos ao esquema rígido de base. Desta forma somos levados a inventar uma função de tipo 2. O PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR envolve um pouco de descoberta mas, em geral, é fácil de descobrir esse ponto de partida pois os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Exemplos de funções de tipo 2: maxList, fall, fib.

Mais exemplos de funções de tipo 2:

```
halfHalf: reparte os elementos duma lista por duas listas, alternadamente
     (esta versão é programada com base na ideia de processar os elementos da lista dois a
dois)
    let rec halfHalf l =
        match l with
            [] -> ([],[])
          | [x] -> ([x],[])
          | x::y::xs ->
                let (us,vs) = halfHalf xs in
                    (x::us, y::vs)
    ;;

addEvenPos: soma todos os elementos duma lista que estão em posições de índice par (0, 2, 4,
...)
    let rec addEvenPos l =
```

```
    match l with
        [] -> 0
      | [x] -> x
      | x::_::xs ->
            x + addEvenPos xs
;;
```

Funções de tipo 3

As **funções de tipo 3** são programadas usando o método indutivo e caracterizam-se pelas seguinte propriedade:

- A chamada recursiva envolve expressões que *não são* subpadrões dos padrões que representam os argumentos da função. As expressões que constituem os argumentos da chamada recursiva são *calculados*.

Orientação prática: As funções de tipo 3 surgem geralmente quando se tenta resolver um problema aplicando uma estratégia previamente pensada, em vez de se tentar encontrar uma solução baseada nos esquemas predefinidos que caracterizam as funções de tipo 1.

Precaução: Programar uma função de tipo 3 em vez duma de tipo 1 ou 2 significa, quase sempre, **complicar desnecessariamente**. Além disso, as funções de tipo 3 são mais difíceis de perceber do que as funções de tipo 1 ou 2. Em todo o caso, ocasionalmente surge uma boa razão para se escrever uma função de tipo 3: por exemplo, o aumento da eficiência (se for caso disso, porque por vezes a função fica menos eficiente).

Exemplo de função de tipo 3:

```
quickSort: ordena lista eficientemente (reduz-se o tratamento de
             uma lista ao tratamento de duas secções dessa lista)
    let rec partition p l =
        match l with
            [] -> ([],[])
          | x::xs ->
              let (a,b) = partition p xs in
                  if x <= p then (x::a,b) else (a, x::b)
    ;;
    let rec quickSort l =
        match l with
            [] -> []
          | x::xs ->
              let (us,vs) = partition x xs in
                  (quickSort us) @ [x] @ (quickSort vs)
    ;;

minSort: ordena lista usando o algoritmo de selecção directa
    let rec removeFromList v l =
        match l with
            [] -> []
          | x::xs ->
                if x = v then xs
                else x::removeFromList v xs
    ;;
    let rec minList l =
        match l with
            [x] -> x
          | x::xs -> min x (minList xs)
    ;;
    let rec minSort l =
        match l with
            [] -> []
          | list ->
              let m = minList list in
                  m::minSort (removeFromList m list)
    ;;
```

A função `quickSort` permite ganhar eficiência. Já a função `minSort` é muito mais complicada do que a função de tipo 1 `sortList` e não é mais eficiente.

Funções de tipo 4

Uma **função de tipo 4** é uma função não recursiva que se define à custa de funções auxiliares de tipo 1, 2, ou 3.

Orientação prática: Existem problemas que não podem ser resolvidos directamente usando o método indutivo. Nestes casos surge a necessidade de escrever uma função de tipo 4. Quase sempre é preciso inventar um novo problema que já se consiga resolver usando o método indutivo e que ajude a resolver o problema original.

Exemplos de funções de tipo 4:

prime: determina de um inteiro é ou não primo

```
    let rec hasDiv n a z =
        if a > z  then false
        else (n mod a = 0) or (hasDiv n (a+1) z)
    ;;
    let prime n =
       not (hasDiv n 2 (n-1))
    ;;
```

width: determina a largura duma árvore binária

```
    type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;

    let rec maxList l =
        match l with
            [x] -> x
          | x::xs -> max x (maxList xs)
    ;;
    let rec sumLists l1 l2 =
        match l1,l2 with
            [],l -> l
          | l,[] -> l
          | x::xs, y::ys -> (x+y)::sumLists xs ys
    ;;
    let rec levels t =
        match t with
            Nil -> []
          | Node(x,l,r) -> 1::sumLists (levels l) (levels r)
    ;;
    let width t =
         if t = Nil then 0
         else maxList (levels t) ;;
```

Eficiência - otimização da última chamada (tail call optimization)

Iteração e recursão em Java

Esqueçamos agora por alguns momentos a linguagem OCaml. A discussão que se segue é feita no contexto da linguagem Java.

Pode obter-se **repetição** em Java de duas formas: usando **iteração** ou usando **recursão**.

O seguinte método calcula o somatório `1 + 1 + 1 + ...` n vezes usando iteração:

```
      int count(int n) {
```

```
        int a = 0 ;
        for( int i = 0 ; i < n ; i++ )
            a++ ;
        return a ;
    }
```

O seguinte método calcula o mesmo resultado, mas agora usando recursão:

```
    int countR(int n) {
        if( n == 0 ) return 0 ;
        else return 1 + countR(n-1) ;
    }
```

Vamos comparar a eficiência das duas funções:

- Do ponto de vista da **velocidade de execução**, ambas as funções têm complexidade linear O(n). Contudo, há um fator constante muito grande a separar a eficiência das funções. Na verdade, em Java, a versão recursiva tende a ser 10 vezes mais lenta do que a segunda por causa das complicações envolvidas na implementação da recursividade.
- Do ponto de vista do **uso da memória**, a versão recursiva perde de forma ainda mais evidente: enquanto a versão iterativa tem complexidade constante O(1), a versão recursiva tem complexidade linear O(n). Realmente, cada chamada recursiva implica sempre algum gasto de memória (criação dum *registo de ativação* na chamada *pilha de execução*) e acontece até que, para argumentos grandes, a execução da função recursiva aborta por falta de memória.

Em Java nunca há qualquer hesitação: se um problema pode ser resolvido de forma simples usando iteração, não há qualquer razão para usar recursão.

Recursão em OCaml e outras linguagens funcionais

Nas linguagens funcionais, a questão anteriormente discutida é muito relevante, porque a filosofia dessas linguagens só admite a utilização de recursão.

Por exemplo, reescrevendo a função anterior em OCaml e testando, é fácil verificar que a sua execução também aborta para argumentos muito grandes.

```
    let rec countR n =
        if n = 0 then 0
        else 1 + countR (n-1)
    ;;
```

Otimização da última chamada

Para minorar o preço a pagar pela recursão, os implementadores de linguagens funcionais descobriram uma técnica de implementação que dá bons resultados, embora só possa ser aplicada numa situação muito particular.

Trata-se da técnica da **otimização da última chamada**. Como o nome indica, esta técnica pode ser aplicada na última chamada que é feita dentro duma função, antes de retornar. A última chamada é tratada assim:

- Em vez de se criar um novo registo de ativação para tratar a última chamada, o que faz é reaproveitar o registo de ativação da própria função e simplesmente saltar para a função chamada (depois de se passarem os argumentos, claro). Repare que nesta situação não é necessário guardar qualquer *endereço de retorno*, e por isso este esquema funciona.

Portanto, a última chamada pode ser implementada sem gastar memória e poupando algum tempo de execução.

O OCaml e a generalidade das linguagens funcionais implementam esta técnica. O Java não a implementa porque não precisa - já dispõe de iteração.

Mas repare que a função OCaml anterior beneficia muito pouco desta técnica de implementação. A última chamada efetuada é uma soma. O que convinha era que a última chamada fosse a chamada recursiva, para se poder executar a função sem gastar memória.

Bem, isso pode fazer-se. Basta reescrever a função como se mostra abaixo, usando uma função auxiliar com um argumento suplementar que serve para acumular o resultado final:

```
let rec countX n a =
    if n = 0 then a
    else countX (n-1) (a+1)
;;
let count n =
    countX n 0
;;
```

Experimente! Por maior que seja o argumento passado para a função `count`, a sua execução nunca aborta por falta de memória.

Preço a pagar

Infelizmente a função `count` é muito menos legível do que a original. A parte pior é ter um sabor imperativo, no sentido em que para perceber o que a função faz é preciso executá-la mentalmente.

Será que, nas linguagens funcionais, somos forçados a escrever este tipo de código em programas reais, que lidam com valores grandes, listas grandes, etc.?

A resposta é "sim", mas só se os argumentos forem mesmo muito grandes. Para ver as magnitudes envolvidas, façamos a experiência de compilar o seguinte programa usando os compiladores `ocamlc` e `ocamlopt`:

```
let rec countR n =
    if n = 0 then 0
    else 1 + countR (n-1)
;;

let rec test n =
    print_int (countR  n) ;
    print_string "\n" ;
    flush stdout ;
    test (2 * n)
;;

test 100000 ;;
```

Correndo o programa gerado com o `ocamlc`, obtém-se:

```
100000
200000
Fatal error: exception Stack_overflow
```

Correndo o programa gerado com o `ocamlopt`, obtém-se:

```
100000
200000
400000
800000
1600000
Fatal error: exception Stack_overflow
```

Algumas funções de ordem superior sobre listas

Num programa que manipule listas, muitas vezes há vantagem algumas destas funções.

Programar bem numa linguagem funcional envolve duas coisas:

- Saber usar bem o método indutivo;
- Sabem reconhecer oportunidades de utilização das funções de biblioteca mais importantes.

Função `map`

Aplica uma função `f: 'a -> 'b` a todos os elementos duma lista, para produzir a lista das imagens.

A função `f` define uma relação de um-para-um, pelo que a lista dos resultados tem o mesmo comprimento da lista de entrada.

```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
let rec map f l =
    match l with
        [] -> []
      | x::xs -> (f x) :: map f xs
;;
```

Esta função está disponível na biblioteca do OCaml como **`List.map`**.

Função `flatMap`

Aplica uma função `f: 'a -> 'b list` a todos os elementos duma lista, para produzir a lista das imagens.

A função `f` define uma relação de um-para-n, pelo que a lista dos resultados tem um comprimento sem qualquer relação com comprimento da lista de entrada.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)
let rec flatMap f l =
    match l with
        [] -> []
      | x::xs -> (f x) @flatMap f xs
;;
```

Esta função não está diretamente disponível na biblioteca do OCaml, pode ser obtida como a combinação de **`List.flatten`** com **`List.map`**. Concretamente, a seguinte definição também é válida.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)
let rec flatMap f l =
    List.flatten (List.map f l)
;;
```

Função `for_all`

Testa se todos os elementos duma lista satisfazem um dado predicado `p: 'a -> bool`.

```
(* for_all : ('a -> bool) -> 'a list -> bool *)
let rec for_all f l =
    match l with
        [] -> true
      | x::xs -> (f x) && for_all f xs
;;
```

Esta função está disponível na biblioteca do OCaml como **`List.for_all`**.

Função `count_all`

Conta todos os elementos duma lista satisfazem um dado predicado `p: 'a -> bool`.

```
(* count_all : ('a -> bool) -> 'a list -> int *)
let rec count_all f l =
    match l with
        [] -> 0
      | x::xs -> (if f x then 1 else 0) + count_all f xs
;;
```

Esta função não está disponível na biblioteca do OCaml.

Mais funções de ordem superior sobre listas

`List.exists` `: ('a -> bool) -> 'a list -> bool` (* Testa se pelo menos um dos valores duma lista satisfaz um dado predicado. *)

`List.filter` : `('a -> bool) -> 'a list -> 'a list` (* Seleciona os elementos duma lista que satisfazem um dado predicado. *)

Função de teste de pertença a uma lista ("membership")

`List.mem` : `'a -> 'a list -> bool` (* Testa se um valor pertence a uma lista. *)

Problemas envolvendo quantificação universal

Verificar se todos os elementos duma lista são pares:

```
let allEven l =
    List.for_all (fun x -> x mod 2 = 0) l
;;
```

Testar se todos os elementos duma lista são menores do que os elementos de outra lista:

```
let allLess l1 l2 =
    List.for_all (fun x ->
          List.for_all (fun y -> x < y) l2) l1
;;
```

E se forem três listas?

```
let allLess3 l1 l2 l3 =
    List.for_all (fun x ->
          List.for_all (fun y ->
             List.for_all (fun z -> x < y && y < z) l3) l2) l1
;;
```

Problemas sobre tabuleiros do jogo das damas

Algumas funções básicas

```
let boardSize = 8 ;;
let maxPos = boardSize - 1 ;;

let direction player =
    if player = black then 1
    else if player = white then -1
    else raise (Arg.Bad "playerToDelta")
;;

let inside (li,co) =
     0 <= li && li <= maxPos && 0 <= co && co  <= maxPos
;;

let nextPos direction (li,co) =
    List.filter inside [(li+direction,co-1); (li+direction,co+1)]
;;
```

Atenção: na função anterior foi corrigida.

Gerar todas as posições dum tabuleiro

```
let rec range a b =
    if a > b then []
    else a::range (a+1) b
;;
let allIdx =
    range 0 maxPos
;;
let allPos =
    List.flatten (List.map (fun li -> List.map (fun co -> (li,co)) allIdx) allIdx)
;;
```

Selecionar todas as posições dum tabuleiro que verificam uma determinada propriedade

```
let filterBoard f board  =
    List.flatten (List.map (fun p -> if f board p then [p] else []) allPos)
;;
```

Exemplo de utilização: todas as posições com uma peça da cor `player`:

```
let playerPos player board  =
    filterBoard (fun b (li,co) -> get b li co = player) board
;;
```