

Universidade da Beira Interior

Programar em OCaml

Introdução pela prática

Simão Melo de Sousa

Aula 2

- conjunto de Mandelbrot
- desenhar curvas
- copiar um ficheiro
- inverter as linhas de um texto
- conversões de inteiros para uma base arbitrária
- conclusão. Quer saber mais?

Conjunto de Mandelbrot

noções por introduzir neste exemplo

- declarações de função
- funções recursivas
- funções de primeira classe, funções anónimas
- funções de ordem superior
- aplicação parcial
- fecho de funções
- recursividade terminal
- call-stack e chamadas terminais
- recursão vs. iteração

```
open Graphics

let width = 800
let height = 800
let k = 100

let norm2 x y = x *. x +. y *. y

let mandelbrot a b =
  let rec mandel_rec x y i =
    if i = k || norm2 x y > 4.
    then i = k
    else
      let x' = x *. x -. y *. y +. a in
      let y' = 2. *. x *. y +. b in
      mandel_rec x' y' (i + 1)
  in
  mandel_rec 0. 0. 0
```

```
let draw () =  
  for w = 0 to width - 1 do  
    for h = 0 to height - 1 do  
      let a = 4. *. float w /. float width -. 2. in  
      let b = 4. *. float h /. float height -. 2. in  
      if mandelbrot a b then plot w h  
    done  
  done  
  
let () =  
  let dim = Printf.sprintf " %dx%d" width height in  
  open_graph dim;  
  draw ();  
  ignore (read_key ())
```

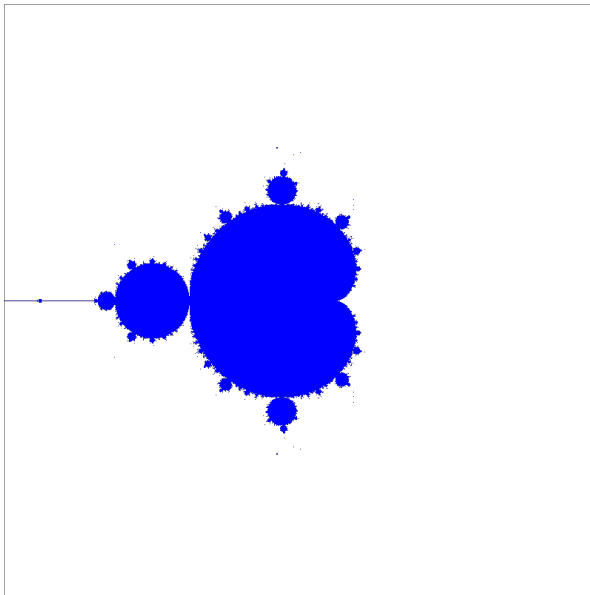
o conjunto de Mandelbrot (link wikipédia aqui) define o conjunto dos pontos (a, b) do plano tais que nenhuma das duas sequências recursivas seguintes tende para o infinito (em valor absoluto)

$$\begin{cases} x_0 &= 0 \\ y_0 &= 0 \\ x_{n+1} &= x_n^2 - y_n^2 + a \\ y_{n+1} &= 2x_n y_n + b \end{cases}$$

mesmo se não há métodos exactos para determinar esta condição, pode ser demonstrado que estas sequências tendem para o infinito logo que $x_n^2 + y_n^2 > 4$ assim

- os pontos do conjunto pertencem ao círculo centrado em $(0, 0)$ e de raio 2
- pode-se definir uma aproximação em que os pontos (a, b) são tais que $x_n^2 + y_n^2 \leq 4$ para os k primeiros termos desta sequência

a precisão desta aproximação depende de k




```
open Graphics

let width = 800
let height = 800
let k = 100

let norm2 x y = x *. x +. y *. y
```

vamos usar o módulo Graphics

e definimos uma janela gráfica com as dimensões 800×800

definimos igualmente limite da nossa aproximação, propondo que k seja igual a 100

este k indica quantos valores da sequência pretendemos calcular

finalmente definimos uma função auxiliar *norm2* que permite o cálculo de $x^2 + y^2$

```
let mandelbrot a b =  
  let rec mandel_rec x y i =  
    if i = k || norm2 x y > 4.  
    then i = k  
    else  
      let x' = x *. x -. y *. y +. a in  
      let y' = 2. *. x *. y +. b in  
      mandel_rec x' y' (i + 1)  
  in  
  mandel_rec 0. 0. 0
```

declaramos a função *mandelbrot* que aceita em parâmetro dois reais *a* e *b*

(desafio: como sabemos que são reais?)

esta função declara uma função recursiva local *mandel_rec* (construção *let-rec-in*)

as funções, como já sabemos, são como qualquer outro valor, podem ser declaradas localmente (como qualquer outra variável local)

```

let mandelbrot a b =
  let rec mandel_rec x y i =
    if i = k || norm2 x y > 4.
    then i = k
    else
      let x' = x *. x -. y *. y +. a in
      let y' = 2. *. x *. y +. b in
      mandel_rec x' y' (i + 1)
  in
  mandel_rec 0. 0. 0

```

na chamada à função *mandel_rec* *x y i*, os parâmetros *x* e *y* são os *i*-ésimos valor da sequência (i.e x_i e y_i)

o calculo em *mandel_rec* prossegue da seguinte forma: começamos por verificar se atingimos a *k*-ésima elemento da sequência ou se quebramos a condição de saída ($x_i^2 + y_i^2 > 4$)

se paramos, então enviamos o booleano que determina se o ponto está dentro ou fora do conjunto (o teste $i = k$)

```

let mandelbrot a b =
  let rec mandel_rec x y i =
    if i = k || norm2 x y > 4. then i = k
    else let x' = x *. x -. y *. y +. a in
         let y' = 2. *. x *. y +. b in
         mandel_rec x' y' (i + 1)
  in mandel_rec 0. 0. 0

```

se a condição de paragem não é verificada, então calcula-se os valores seguintes da sequência (x' e y')

e recursivamente chamamos $mandel_rec\ x'\ y'\ (i + 1)$ que tratará de testar o fim ou de calcular o ponto seguinte da sequência

note a utilização de a e de b (que não são passados em parâmetro): são visíveis via a função *mandelbrot*

a chamada inicial, o corpo de *mandelbrot*, é $mandel_rec\ 0.\ 0.\ 0$ (o valor inicial de x e de y e a indicação de que é a iteração 0)

```
let draw () =  
  for w = 0 to width - 1 do  
    for h = 0 to height - 1 do  
      let a = 4. *. float w /. float width -. 2. in  
      let b = 4. *. float h /. float height -. 2. in  
      if mandelbrot a b then plot w h  
    done  
  done
```

para desenhar cada ponto o conjunto (tendo em conta a precisão k) basta então varrer cada linha e cada coluna da janela gráfica e desenhar todos os píxeis (a, b) que são assinalados como *true* pela função *mandelbrot*

é o objectivo da função *draw*

a declaração `let draw () = ...` indica que se trata de uma função e que esta não precisa de argumento particular

as variáveis (contadores de ciclos) w e h permitam percorrer cada ponto da janela gráfica

para desenhar o conjunto de Mandelbrot numa janela $width \times height$ para pontos (a, b) em $[-2, 2] \times [-2, 2]$, definimos duas variáveis locais a e b da forma

```
let a = 4. *. float w /. float width -. 2. in  
let b = 4. *. float h /. float height -. 2. in
```

se *mandelbrot a b* devolver *true* então este ponto pertence ao conjunto e podemos desenhá-lo (recorremos à função *plot*)

a função *plot x y* do módulo *Graphics* desenha o ponto (x, y)

```
if mandelbrot a b then plot w h
```

```
let () =  
  let dim = Printf.sprintf " %dx%d" width height in  
  open_graph dim;  
  draw ();  
  ignore (read_key ())
```

finalmente temos os ingredientes todos para desenhar o conjunto

definimos a dimensão *dim* da janela por abrir a custa da função `sprintf` que devolve uma string conforme o padrão dado em parâmetro

esta string é dada à função `open_graph`

invocamos depois a função `draw` e permitimos que o utilizador prima uma tecla para assinalar o fim da execução

as funções podem ser globais ou locais, como qualquer outra declaração

```
# let f x = x + 1 ;;
val f : int -> int = <fun>
# f 5 ;;
- : int = 6
# let sqr x = x * x in sqr 3 + sqr 4 ;;
- : int = 25
```

em particular, como são expressões quaisquer, tem um valor, podem ser aninhadas em qualquer outra expressão e respeitam as habituais regras de porte

```
# (let sqr x = x * x in sqr 3) + sqr 4 ;;
Error: Unbound value sqr
Hint: Did you mean sqrt?
```

por omissão não são recursivas, pelo que é necessário indicá-las explicitamente com *rec*

```
# let fact x = if x = 0 then 1 else x * fact (x-1);;
Error: Unbound value fact
# let rec fact x = if x = 0 then 1 else x * fact (x-1);;
val fact : int -> int = <fun>
# fact 12 ;;
- : int = 479001600
```


este caso é um caso subtil:

```
# let g x = x * 2;;  
val g : int -> int = <fun>  
# let g y = if y = 0 then 5 else 1 + g (y - 1);;  
val g : int -> int = <fun>  
# g 5 ;;  
- : int = 9
```

não se engane, a segunda função g não é recursiva, **ela refere-se à função g anterior !!**

é uma situação possível e não um erro

é a razão pela qual é complicado o compilador adivinhar **sempre** a recursividade — é preciso o programador dar esta informação

```
# let rec g y = if y = 0 then 5 else 1 + g (y - 1);;  
val g : int -> int = <fun>  
# g 5;;  
- : int = 10
```

na verdade, a sintaxe concreta para definir funções é introduzida pela palavra chave **function**

é a palavra chave OCaml para o λ do cálculo λ de que já falamos

```
function x -> x + 1
```

significa o valor que é uma **função que a um x associa $x + 1$**

por isso é inferido que é uma função de inteiro para inteiro (porque a x é somado 1)

tal função é designada de **função anónima**

porque não tem nome: é realmente um valor... o valor **função sucessor**, assim

```
let succ x = x + 1
```

é exactamente equivalente (na verdade é açúcar sintático de...)

```
let succ = function x -> x +1
```

...sintacticamente igual a qualquer outra declaração de variável!!!

as funções anónimas representam a essência do que são funções

e, claro, podem ser aplicados a argumentos

```
# (function x -> x + 5) 10;;  
- : int = 15  
# (function x -> function y -> x * y);;  
- : int -> int -> int = <fun>  
# (function x -> function y -> x * y) 5 7;;  
- : int = 35
```

a palavra chave **function** introduz funções com um só parâmetro

se pretendemos uma função binária (ou até mesmo n -ária), recorremos ao aninhamento da construção **function**

múltiplos argumento via aninhamento de `function`: possível, mas pouco prático

daí a construção `fun` que é açúcar sintáctico para a composição sucessiva de `function`

```
# (function x -> function y -> function z -> function t -> x + y * z + t);;  
- : int -> int -> int -> int -> int = <fun>  
# fun x y z t -> x + y * z + t;;  
- : int -> int -> int -> int -> int = <fun>  
# (fun x y z t -> x + y * z + t) 2 3 4 5;;  
- : int = 19
```

vamos mostrar a origem da notação dos tipos das funções (como em $int \rightarrow int \rightarrow int$), surpreendentemente a declaração

```
# (function x -> function y -> x * y) 5;;
- : int -> int = <fun>
```

não dá erro

até diz que é uma função de int para int !!

estudemos mais em detalhe este fenómeno:

```
# let f = (function x -> function y -> x * y) 5;;
val f : int -> int = <fun>
# f 7;;
- : int = 35
```

a função f é na verdade a função $function y \rightarrow x * y$ em que x foi instanciado por 5, ou seja

$$f \triangleq function y \rightarrow 5 * y$$

a explicação deste fenómeno está ligada, mais uma vez as famosas regras de ouro de OCaml, assim

```
function x -> function y -> x + y
```

é precisamente o **valor** seguinte:

a função que a x associa uma função $function\ y \rightarrow x + y$

onde $function\ y \rightarrow x + y$ é uma função que a um y associa o valor $x + y$

assim o tipo global é: $int \rightarrow (int \rightarrow int)$

esta função é uma função que tendo x devolve uma função (como valor de retorno)!!!

o que, se assumirmos a associatividade direita de \rightarrow , nos dá
 $int \rightarrow int \rightarrow int$

Está explicada a notação do tipo das funções... faz sentido

funções e o conceito de fecho

OCaml, no momento da definição de uma função (digamos f), define (**calcula**) internamente o seu valor que será posteriormente sempre usado quando invocada

chamamos ao resultado deste calculo o **fecho da função f**
de forma resumida, o valor da função é calculada de forma a que este não dependa mais do ambiente onde foi criado, assim

```
let a = 5
let f x = x + 2 * a
```

atribuí à f a função que a x **associa o valor $x + 10$** e não o valor $x + 2 * a$ em que a vale 5

vejamos uma consequência imediata

```
# let a = 5;;
val a : int = 5
# let f x = x + 2 * a ;;
val f : int -> int = <fun>
# f 3;;
- : int = 13
```

```
# let a = 10;;
val a : int = 10
# f 3;;
- : int = 13
```

f não depende de a ! copiou para o seu fecho todos os valores dos quais depende

vamos estudar um uso interessante da noção de fecho

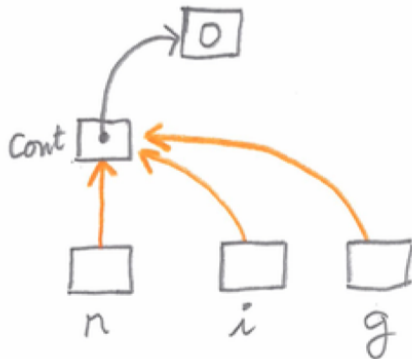
considere as seguintes definições OCaml

```
# let reset,incr,get =  
    let cont = ref 0 in  
    let r () = cont := 0 in  
    let i () = cont := !cont + 1 in  
    let g () = !cont in  
        r,i,g;;  
val reset : unit -> unit = <fun>  
val incr : unit -> unit = <fun>  
val get : unit -> int = <fun>
```

```
# !cont;;  
Error: Unbound value cont  
# get ();;  
- : int = 0  
# incr ();;  
- : unit = ()  
# get ();;  
- : int = 1  
# incr ();incr (); get ();;  
- : int = 3  
# reset (); get ();;  
- : int = 0
```

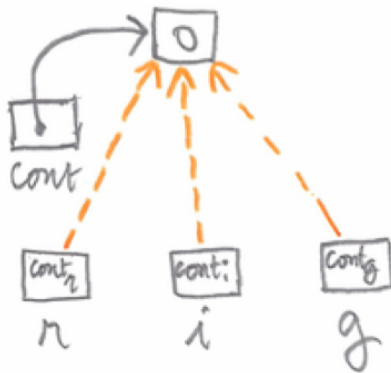

fechos, o que acontece?

```
let reset, incr, get =  
  let cont = ref 0 in  
  let r () = ... cont ... in  
  let i () = ... cont ... in  
  let g () = ... cont ... in  
  r, i, g
```



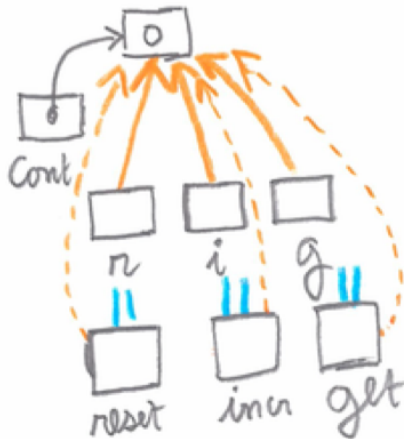
fechos, o que acontece?

```
let reset, incr, get =  
  let cont = ref 0 in  
  let r () = ... cont ... in  
  let i () = ... cont ... in  
  let g () = ... cont ... in  
  r, i, g
```



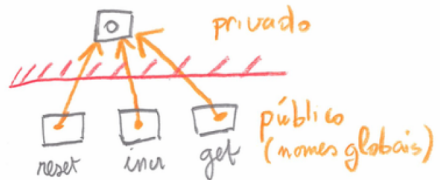
fechos, o que acontece?

```
let reset, incr, get =  
  let cont = ref 0 in  
  let r () = ... cont... in  
  let i () = ... cont... in  
  let g () = ... cont... in  
  r, i, g
```



fechos, o que acontece?

```
let reset, incr, get =  
  let cont = ref 0 in  
  let r () = ... cont ... in  
  let i () = ... cont ... in  
  let g () = ... cont ... in  
  r, i, g
```



esta técnica permite usar para o seu proveito o mecanismo do fecho das funções fornecendo um meio prático de encapsulamento de dados (a referência `cont` neste caso)

`cont`, `r`, `i` e `g` são variáveis locais

`cont`, em particular, é copiado nos fechos das funções locais (cada função tem uma referência a apontar para o valor apontado por `cont`)

por seu turno, as funções globais `reset`, `incr` e `get` são inicializadas com as funções identificadas localmente por `r`, `i` e `g`

o fecho destas funções globais contém assim uma cópia da referência `cont`

no fim da declaração das funções globais, os identificadores locais desaparecem

e assim o valor inteiro acessível por `cont`, **só o é daí em diante** por estas três funções globais

vimos que as funções em OCaml podem naturalmente devolver funções, mas podem receber em parâmetro?

a resposta, sem surpresas, é sim... as funções são valores como quaisquer outros

vejamos alguns exemplos artificiais, mas simples e elucidativos

```
# let rec func b n =  
    if b then if n <= 1 then 1  
               else (func true (n-1)) + (func true (n-2))  
    else if n <= 0 then 1 else n * (func false (n-1));;  
val func : bool -> int -> int = <fun>  
# let misterio1 = func true;;  
val misterio1 : int -> int = <fun>  
# let misterio2 n = func false n;;  
val misterio2 : int -> int = <fun>
```

o que são as funções *misterio1* e *misterio2*? e essa?

```
# let what_function_is_this b = if b then (fun x -> 2 * x)  
                                else (fun y -> y * y);;  
val what_function_is_this : bool -> int -> int = <fun>
```

um aproveitamento sintático interessante da avaliação parcial é o seguinte

```
# let f n m = n + 2 * m;;  
val f : int -> int -> int = <fun>  
# let g m = f 5 m;;  
val g : int -> int = <fun>
```

podemos querer definir a função unária g como sendo a função binária f especializada para o valor 5 como primeiro argumento, ficando o segundo argumento proveniente do argumento de g

mas já que podemos devolver funções, podemos simplificar a definição directamente para

```
# let g = f 5;;  
val g : int -> int = <fun>
```

g é a função que é devolvida por f 5

as duas variantes de g são absolutamente idênticas,
mas a segunda é **mais elegante**

em certas situações é prático poder definir várias funções cujas definições dependem umas das outras: são **funções mutuamente recursivas**

```
let rec f1 ... = ....f1...f2...  
and f2  ... = ....f1...f2...  
and ...
```

as f_i funções assim definidas podem referir-se umas as outras naturalmente
um exemplo: as sequências masculinas e femininas de Hofstadter

$$\begin{aligned} F(0) &= 1 \\ M(0) &= 0 \\ F(n) &= n - M(F(n-1)), \quad n > 0 \\ M(n) &= n - F(M(n-1)), \quad n > 0 \end{aligned}$$

```
let rec f = function  
  | 0 -> 1  
  | n -> n - m(f(n-1))  
and m = function  
  | 0 -> 0  
  | n -> n - f(m(n-1))
```


observemos novamente a função factorial, mais particularmente a sua execução

```
let rec fact = function  
  | 0 -> 1  
  | n -> n * fact (n-1)
```

$$\text{fact } 6 = 6 \times \text{fact } 5$$

$$\text{fact } 6 = 6 \times 5 \times \text{fact } 4$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times \text{fact } 3$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 3 \times \text{fact } 2$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 3 \times 2 \times \text{fact } 1$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times \text{fact } 0$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 3 \times 2$$

$$\text{fact } 6 = 6 \times 5 \times 4 \times 6$$

$$\text{fact } 6 = 6 \times 5 \times 24$$

$$\text{fact } 6 = 6 \times 120$$

$$\text{fact } 6 = 720$$

no computador, a mecânica de execução de tal função recursiva imita perfeitamente o efeito **pirâmide** que aqui presenciamos

este efeito tem lugar na **pilha de chamada** (*call stack*)

na pilha de chamadas são colocadas as funções e procedimentos em actual execução

cada função em execução tem ali toda a informação necessária a sua execução na máquina de suporte

assim

numa chamada de função temos sempre dois actores: quem chama a função (digamos g , designado de **caller**) e a própria função chamada (digamos f , designada de **callee**)

quando a execução está a processar g , este está no topo da pilha

quando g chama f , é alocado espaço na pilha de chamadas para o ambiente de execução de f

f executa-se, e g – que está na pilha a seguir a f , fica a espera que f termine, devolva o seu resultado e seja removido do topo da pilha, para prosseguir com a sua execução

pilha de chamadas para *fact* 6

quando *fact* 6 é invocada

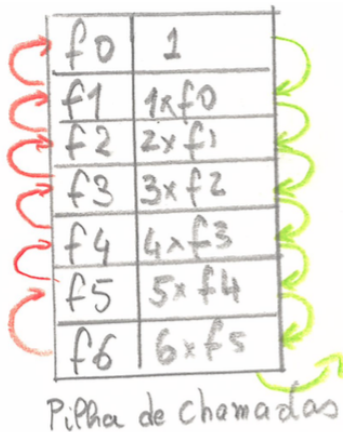
esta calcula $6 * \text{fact } 5$

a multiplicação só consegue ser calculada quando se souber o resultado de *fact* 5

fact 5 é invocado (empilhado na *call stack*)

esta calcula $5 * \text{fact } 4$

a multiplicação só consegue ser calculada quando se souber o resultado de *fact* 4, **etc.**



sem os devidos cuidados, uma função recursiva pode causar esse erro (exceção)

Stack overflow during evaluation (looping recursion?).

o número de chamadas recursivas causou um empilhamento num número demasiado grande na call stack (que apesar de ter um tamanho confortável, é limitado)

como a atual execução obriga a um uso abusivo da *call stack*, uma exceção é levantada

como é uma causa comum, o OCaml suspeita que possa ser uma recursão *infinita*

mas haverá forma de evitar o efeito pirâmide na recursão?

a resposta é **sim**

a função `mandel_rec` é um exemplo de função recursiva sem este efeito: é dita **recursiva terminal**

para qualquer função recursiva (por extensão qualquer processo iterativo) existe uma função recursiva terminal equivalente

pode, no entanto, não ser fácil encontrar/definir tal função

estudemos o exemplo da factorial

o efeito pirâmide tem origem na avaliação da chamada recursiva $i \times (i - 1)!$
a multiplicação só pode ser calculada apos o calculo de $(i - 1)!$, daí ser posta em espera enquanto se calcula esta última (e recursivamente...)

haverá forma de contornar a espera?

sim: transpor os cálculos por fazer para os parâmetros da função de forma a que cada cálculo possa ser realizado na **passagem de parâmetro**

```
let rec fact_fast n acc = if n < 0 then invalid_arg "argumento negativo"
                          else if n < 2 then acc
                          else fact_fast (n-1) (acc * n)

let fact n = fact_fast n 1
```

```
fact 4 = fact_fast 4 1
      = fact_fast 3 4
      = fact_fast 2 12
      = fact_fast 1 24
      = 24
```

```
let rec fact_fast n acc = if n < 0 then invalid_arg "argumento negativo"
                          else if n < 2 then acc
                          else fact_fast (n-1) (acc * n)

let fact n = fact_fast n 1
```

de notar que em cada chamada recursiva nenhum cálculo fica pendente: $n - 1$ e $acc * n$ podem ser avaliados de imediato (todos os valores intermédios, 1, n e acc , são conhecidos)

de notar igualmente a semelhança desta função recursiva terminal com a versão iterativa

```
# let fact_iter n =
  let acc = ref 1 in
  for i = 1 to n do
    acc := !acc * i
  done; !acc;;
val fact_iter : int -> int = <fun>
# fact_iter 4;;
- : int = 24
```

de forma semelhante, apresentamos 3 versões da função fibonacci: a natural, iterativa e recursiva terminal

$$\text{fib } n \triangleq \begin{cases} 1 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{se } n > 1 \end{cases}$$

```
let rec fib = function
  | 0 -> 1
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2)
```

```
let fib_iter n =
  if (n<0) then invalid_arg 'negativo'
  else
    if n < 2
    then 1
    else let a = ref 1
         and b = ref 1 in
         for num = 2 to n do
           let temp = !a in
           a := !a + !b;
           b := temp
         done;
    !a
```

```
let fib n =
  let rec fib_rec_term n a b =
    match n with
    | 0 | 1 -> a
    | _ -> fib_rec_term (n-1) (a+b) a
  in
  fib_rec_term n 1 1
```

como referido, a versão recursiva terminal tem semelhanças com a versão iterativa

as funções recursivas terminais são **elegantes e muito eficientes**, tem performances semelhantes às versões iterativas

a compilação de funções recursivas terminais e das suas equivalentes iterativas **resultam no mesmo código de baixo nível**

compiladores podem até tirar proveito da recursividade terminal

mas então, que estilo preferir?

já abordamos esta questão no contexto da sequência, é uma questão de estilo de programação

embora a recursão, como já referido, tem a sua elegância própria e o seu suporte particular em OCaml

um bom programador sabe tirar proveito da recursividade terminal

um aparte sobre a função fibonacci

a versão recursiva terminal da função fibonacci é bastante eficiente

é, por exemplo, tão eficiente computacionalmente quanto a versão com **memoização** (linear) mas mais eficiente em espaço (constante)

uma função (recursiva) f com **memoização** é uma função a que associamos um dicionário (e.g. vector, lista, tabela de hash, etc.) que arquiva para cada i o valor de $(f\ i)$ a medida que estes são calculados, para evitar novo cálculo mais adiante

a versão de fibonacci com memoização é linear em tempo e em memória

mas existe uma versão recursiva **logarítmica** (logo, **melhor ainda**) do calculo da sequência de fibonacci baseada no produto de matrizes (ver ficha de exercícios)

resta-nos ver uma característica particular do suporte de OCaml às funções, via as funções anônimas, que nos permite caracterizar finalmente e totalmente as funções OCaml como

funções de ordem superior

se uma função pode retornar uma função, será que - por simetria - pode receber funções?

a resposta é sim, porque são valores como quaisquer outros

vamos introduzir e explicar esta característica com base num exemplo simples

pretende-se uma função que calcula a soma de 1 até 10

```
let soma () =  
  let rec soma_aux i = if i > 10 then 0 else i + soma_aux (i+1)  
  in soma_aux 1
```

de 1 até n ?

```
let soma n =  
  let rec soma_aux i n = if i > n then 0 else i + soma_aux (i+1) n  
  in soma_aux 1 n
```

e recursiva terminal?

```
let soma n =  
  let rec soma_aux i n acc =  
    if i > n then acc  
    else soma_aux (i+1) n (acc+i)  
  in soma_aux 1 n 0
```

```
let soma n =  
  let rec soma_aux i n acc = if i > n then acc  
                              else soma_aux (i+1) n (acc+i)  
  in soma_aux 1 n 0
```

e se no lugar da soma, pretendemos o produto (ou seja... a factorial)?

```
let produto n =  
  let rec soma_aux i n acc = if i > n then acc  
                              else soma_aux (i+1) n (acc * i)  
  in produto_aux 1 n 1
```

como generalizar para qualquer operação? digamos f e partindo do valor *init*

```
let f_orio f init n =  
  let rec f_aux f i n acc = if i > n then acc  
                              else f_aux f (i+1) n (f acc i)  
  in f_aux f 1 init n
```

```
# f_orio (+) 0 10;;  
- : int = 55  
# f_orio ( * ) 1 10;;  
- : int = 3628800
```

```
# f_orio (fun a b -> a + 2 * b) 1 10;;  
- : int = 111
```

Desenhar curvas

noções por introduzir neste exemplo

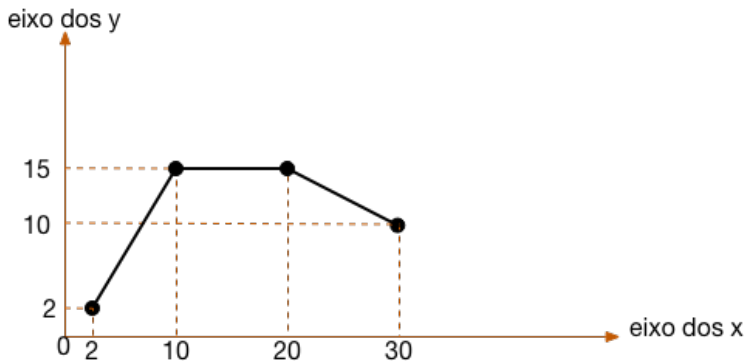
- pares, tuplos
- ordem de avaliação
- definições por filtro, motivo universal
- registos e campos mutáveis

o desafio, desta vez, é desenhar uma curva a partir de um conjunto de pontos lidos da entrada standard

cada ponto é dado pelas suas coordenadas inteiras, isto é, um par de inteiros

vamos usar para este propósito a capacidade de base do OCaml em lidar com tuplos

para as coordenadas $(2, 2)$, $(10, 15)$, $(20, 15)$, $(30, 10)$ a curva pretendida é a seguinte:



```
let n = read_int ()

let read_pair () =
  let x = read_int () in
  let y = read_int () in
  (x, y)

let data = Array.init n (fun i -> read_pair ())

let compare (x1, y1) (x2, y2) = x1 - x2
let () = Array.sort compare data

open Graphics

let () =
  open_graph " 200x200";
  set_line_width 3;
  let (x0,y0) = data.(0) in moveto x0 y0;
  for i = 1 to n-1 do
    let (x,y) = data.(i) in
    lineto x y
  done;
  ignore (read_key ())
```

```
let n = read_int ()

let read_pair () =
  let x = read_int () in
  let y = read_int () in
  (x, y)
```

a função `read_pair` tem por objectivo ler uma coordenada na forma de dois valores inteiros, a partir do `stdin`

em OCaml, valores podem ser agregados num só valor: um tuplo

assim o valor `(x,y)` é um par de inteiros, isto é: **um valor tuplo que é composto por dois valores inteiros** (tipo: `int * int`)

OCaml suporta naturalmente a definição e uso de qualquer tipo de tuplos

`(x,y,z,t)`, por exemplo, é um valor tuplo que agrega 4 valores

se `x` for inteiro, `y` char, `z` float e `t` booleano, então este valor tuplo tem por tipo:

`int * char * float * bool`

```
let data = Array.init n (fun i -> read_pair ())
```

o identificador `data` é definido como o vector de pares inicializado com o recurso à função `Array.init` (à diferença do `Array.make` que já exploramos)

a função `init` permite criar um vector e inicializar individualmente cada célula do vector (com base no seu índice `i`)

literalmente, esta inicialização pode ser lida como:

`data` é um vector de tamanho n em que cada célula i (de 0 a $n - 1$) é inicializada pela função `read_pair`

`Array.init` é uma função de ordem superior.

assim, se o vector é de tamanho 4 e que `Array.init 4 (fun i -> read_pair ())` é invocada,

a introdução da sequência de inteiros 20 15 2 2 30 10 10 15

origina o vector

(20,15)	(2,2)	(30,10)	(10,15)
---------	-------	---------	---------

```
let compare (x1, y1) (x2, y2) = x1 - x2  
let () = Array.sort compare data
```

para desenhar a curva composta pelos pontos arquivados em `data` é preciso ordenar os pontos por ordem crescentes das abcissas

para tal, começamos por definir uma função de comparação própria à ordenação que pretendemos, a função `compare`

os critérios de comparação para usar em ordenação devolvem um inteiro e seguem o seguinte padrão ao comparar digamos a com b

se $a = b$ então devolve 0

se $a < b$ então devolve -1 (valor negativo)

se $a > b$ então devolve 1 (valor positivo)

neste caso concreto, pretendemos ordenar pontos num plano com base na abcissa

para dois pontos (x_1, y_1) e (x_2, y_2) basta saber o resultado de $x_1 - x_2$

aproveitamos o conhecimento de que os dois parâmetros de `compare` são pares e logo na declaração distinguir as suas componentes: é uma **declaração por filtro**

a ordenação do vector é feita **in-place** a custa da função `sort` que precisa de ser instrumentada pelo critério de comparação por usar: `compare`

```
open Graphics

let () =
  open_graph " 200x200";
  set_line_width 3;
  let (x0,y0) = data.(0) in moveto x0 y0;
  for i = 1 to n-1 do
    let (x,y) = data.(i) in
      lineto x y
  done;
  ignore (read_key ())
```

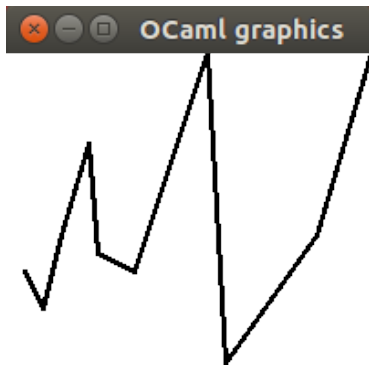
o resto do programa prossegue sem surpresa

após abertura da janela gráfica, coloca-se o ponto activo na primeira coordenada (x_0, y_0) arquivada em *data*

daí em diante, para cada ponto (x_i, y_i) restante de *data* desenha-se um segmento de recta entre o ponto activo e este (com recurso à função *lineto*, que recoloca o ponto activo no seu argumento)

input:

10
20
60
70
80
10
80
50
90
30
100
200
200
120
30
110
200
45
150
170
100



os tuplos são elementos de produtos cartesianos
agrupam assim valores de tipos possivelmente diferentes, mas em posições conhecidas/fixas

```
# ('a',8);;  
- : char * int = ('a', 8)
```

é um par (tuplo de dois elementos) de um caractere e de um inteiro
mas é diferente de (8,'a') (de tipo `int*char`)
tuplos são expressões como qualquer outros, reduzem-se para valores e têm todos um tipo
podem assim ser aninhados ou definidos a custa de outras expressões
para o caso particular dos pares, existem funções de projecção predefinidas *fst* (*first*) e *snd* (*second*)

```
# fst ("ola" ^ " tudo bem", if 4 > 8 then 5=8 else true);;  
- : string = "ola tudo bem"  
# snd ("ola" ^ " tudo bem", if 4 > 8 then 5=8 else true);;  
- : bool = true  
# (1,('c',"ola"));;  
- : int * (char * string) = (1, ('c', "ola"))  
# ((1,'r'),("ola",3.14));;  
- : (int * char) * (string * float) = ((1, 'r'), ("ola", 3.14))
```


a igualdade = é polimórfica, consegue lidar com a estrutura dos tuplos

mas só sabe comparar o que é comparável (i.e. objectos de mesmo tipo)

```
# ((1,'r'),("ola",3.14)) = ((1,'r'),("ola",3.14));;
- : bool = true
# let x = 1;;
val x : int = 1
# ((x,'r'),("ola",3.14)) = ((1,'r'),("ola",3.14));;
- : bool = true
# let r = (1,'r');;
val r : int * char = (1,'r')
# (r,("ola",3.14)) = ((1,'r'),("ola",3.14));;
- : bool = true
# ((1,'r'),("ola",3.14)) = ((1,'r'),(3.14,"ola"));;
```

~~~~~

Error: This expression has type float but an expression was expected  
of type string

mostramos aqui alguns exemplos que demonstram como usar o mecanismo de filtro ao seu proveito no caso dos tuplos (e via a construção `let`)

```
# let (x,y,z)=((1,'r'),('tudo',(2,4)),"ola",3.14));;
val x : int * char = (1, 'r')
val y : string * (int * int) = ("tudo", (2, 4))
val z : string * float = ("ola", 3.14)
# let ((a,b),(c,d),z)=((1,'r'),('tudo',(2,4)),"ola",3.14));;
val a : int = 1
val b : char = 'r'
val c : string = "tudo"
val d : int * int = (2, 4)
val z : string * float = ("ola", 3.14)
# let ((a,b),_,(c,_))=((1,'r'),('tudo',(2,4)),"ola",3.14));;
val a : int = 1
val b : char = 'r'
val c : string = "ola"
```

de notar o uso do `_` para representar o padrão “qualquer coisa”

# complementos sobre tuplos e filtro

as construções por filtro são construções condicionais que se baseiam na forma que o argumento da construção tem

o argumento de decisão é assim **puramente sintático** (a *forma* do objeto): permite a decisão conforme a **estrutura** do argumento

o filtro é um mecanismo primitivo para o qual existem várias sintaxes (são na verdade todos *açucares sintácticos* do mecanismo de base)

basicamente o filtro resume-se informalmente numa sequência de: *se o argumento é desta forma então fazer aquilo* por exemplo

```
# let f = function
  | 0 -> 5
  | 1 -> 9
  | _ -> 4;;
val f : int -> int = <fun>
```

```
# let g x = let (a,b) = x in a+b
val g : int * int -> int = <fun>
```

ou

```
# let g (a,b) = a+b
val g : int * int -> int = <fun>
```

que se lê: seja *f* a função que tem um parâmetro tal que se este for 0 então o resultado é 5, se este for 1 então o resultado é 9 qualquer outro argumento inteiro devolve 4

## complementos sobre tuplos e filtros

```
let soma x y = let (a,b) = x in  
               let (c,d) = y in  
               (a+c,b+d)
```

```
let soma = fun (a,b) (c,d) -> (a+c,b+d)  
  
let soma (a,b) (c,d) = (a+c,b+d)
```

estas três versões equivalentes (de tipo  $int * int \rightarrow int * int \rightarrow int * int$ ) usam variantes do mecanismo de filtro e introduzem todos uma função que soma dois pares de inteiros

ambas construções `let` e `fun` desestruturam os seus argumentos (`x` e `y`) nas diferentes formas em que esses podem tomar, aqui só uma é possível: o par `(_,_)`  
`a,b,c,d` são **variáveis locais** que tomam os valores das componentes dos pares

podemos finalmente explicar melhor a construção `let` seguinte

```
let () = print_string "ola!\n"
```

```
let _ = print_string "ola!\n"
```

avalia-se o valor da expressão `print_string` e verifica-se que é igual ao valor `()` (o único possível do tipo `unit`): **é um filtro inequívoco**

asseguramos assim por tipagem que o resultado esperado é bem `()`  
no segundo caso, simplesmente ignoramos o resultado (`_ = qualquer coisa`)

**em conclusão:** a construção `let`, quando a expressão à esquerda do `=` não é um identificador, **é uma construção por filtro** em que o filtro é **inequívoco**

tendo ao nosso dispor tuplos, podemos escrever funções como

```
# let dupla_soma (x,y) = 2 * (x + y);;
val dupla_soma : int * int -> int
```

ou

```
# let dupla_soma x y = 2 * (x + y);;
val dupla_soma : int -> int -> int
```

qual a diferença?

na verdade, ambas implementam a mesma função

mas a primeira não permite avaliação parcial ao contrário da segunda

```
# dupla_soma (5,4);;
- : int = 18
# dupla_soma (5,_);;
Error: Syntax error: wildcard "_" not expected.
```

ou

```
# dupla_soma 5 4;;
- : int = 18
# dupla_soma 5;;
- : int -> int = <fun>
```

à esquerda, (5,4) é um valor, enquanto à direita 5 4 são dois valores, por isso podemos ter aplicação parcial

diz-se da segunda função que é a versão **curryficada** (em inglês: *curryfied*, em referência à Haskell Curry) da primeira

em termos práticos, a versão curryficada é mais cómoda

o tipo produto cartesiano permite a representação agregada de informação não homogênea (de tipos diferentes)

no entanto, pode ser cómodo dispor de nome para cada componente e de respectivas funções de selecção/projecção

é o que as estruturas da linguagem C fornecem

OCaml tem para esse mesmo efeito registos: são tuplos em que cada componente tem um nome

à diferença dos tuplos que podem ser definidos e usados *on-the-fly*, é necessário introduzir previamente o tipo registo pretendido antes de poder definir e usar os seus elementos

```
# type date = {day : int; month : int; year : int};;  
type date = { day : int; month : int; year : int; }
```

```
# type complex = {re:float; im:float};;
type complex = { re : float; im : float;}
```

vs.

```
#type complexo = float * float;;
type complexo = float * float
```

## definição de registos

```
# let c = {re=2.;im=3.};;
val c : complex = {re = 2.; im = 3.}
```

a ordem dos campos pouco importa, já que são referenciados pelo nome (nos tuplos, a ordem importa!)

```
# let cc = {im=9.2;re=5.9};;
val cc : complex = {re = 5.9; im = 9.2}
```

a construção **with** permite facilitar a definição de registos (a partir de outros, aqui c)

```
# let d = {c with im = c.im +. 8.};;
val d : complex = {re = 2.; im = 11.}
```

note a sintaxe **c.im** que permite o aceso ao valor do campo im do registo c

podemos definir uma função com base em argumentos de tipo registo, acedendo directamente aos diferentes campos pelo nome

```
# let add_complex c1 c2 = {re = c1.re +. c2.re; im = c1.im +. c2.im};;
```

o mecanismo de filtro pode também ser aqui usado

```
val add_complex : complex -> complex -> complex = <fun>
# let mult_complex c1 c2 =
    let {re=x1;im=y1} = c1 in
    let {re=x2;im=y2} = c2 in
    {re=x1*.x2-.y1*.y2 ; im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
```

podemos usar estas funções tendo os complexos *c* e *d* definidos anteriormente

```
# add_complex c d;;
- : complex = {re = 4.; im = 14.}
# mult_complex c d;;
- : complex = {re = -29.; im = 28.}
```



os registos são, mais uma vez, como quaisquer outras expressões em OCaml

tem um valor, tem um tipo e são **imutáveis**

os seus campos também e estes são **imutáveis**

... por omissão, i.e. há uma excepção!!

o campo mutável

```
type conta = {dono : string; mutable saldo : float}
```

campos mutáveis declaram-se com a palavra chave `mutable`

o acesso ao campo é identico ao caso habitual

```
type conta = {dono : string; mutable saldo : float}

exception Operacao_Invalida

let create_account name = {dono=name; saldo=0.0}

let get_owner x = x.dono

let get_balance x = x.saldo
```

a alteração *in-place* do valor de um campo mutável faz-se pelo operador ←

```
let inc_account x v = x.saldo <- x.saldo +. v

let decr_account x v=
  if x.saldo -. v < 0.0
  then raise Operacao_Invalida
  else x.saldo <- x.saldo -. v
```

Podemos finalmente revelar como as referências em OCaml são implementadas

```
type 'a ref = {mutable contents : 'a}

let ref x = {contents = x}

let (:=) r v = r.contents <- v
```

onde uma declaração do estilo `let (ident) x y = ...` (onde `ident` é um identificador constituído exclusivamente de símbolos e não por caracteres) permite a definição de uma função **binária** mas como operador infixo (como a soma `+` por exemplo)

```
# type 'a variavel = {mutable conteudo : 'a};;
type 'a variavel = { mutable conteudo : 'a; }
# let aponta x = {conteudo = x};;          (* no lugar de ref *)
val aponta : 'a -> 'a variavel = <fun>
# let get r v = r.conteudo <- v;;
val get : 'a variavel -> 'a -> unit = <fun>
# let (<==>) = get;;                        (* no lugar de := *)
val ( <==> ) : 'a variavel -> 'a -> unit = <fun>
# let a = aponta 6;;
val a : int variavel = {conteudo = 6}
# a <== 8;;
- : unit = ()
# a;;
- : int variavel = {conteudo = 8}
```

# ordem de avaliação em tuplos e registos: armadilha!

voltamos à ordem de avaliação: não escreva programas que dependem dela!

a ordem de avaliação de componentes de tuplos ou de registos não está especificada nos documentos de referência de OCaml

confiar na ordem por omissão

```
# (read_int () , read_int ());;  
4  
6  
- : int * int = (6, 4)
```

forçar a ordem de avaliação

```
# let x = read_int () in  
  let y = read_int () in  
    (x,y);;  
4  
6  
- : int * int = (4, 6)
```

---

## Copiar um ficheiro

## noções por introduzir neste exemplo

- input/output
- canais
- exceções

o objectivo do programa seguinte é copiar o conteúdo de um ficheiro para outro

os ficheiros fonte e alvo são passados pela linha de comando



```
let copy_file f1 f2 =  
  let c1 = open_in f1 in  
  let c2 = open_out f2 in  
  try  
    while true do output_char c2 (input_char c1) done  
  with End_of_file ->  
    close_in c1; close_out c2  
  
let () = copy_file Sys.argv.(1) Sys.argv.(2)
```

o processo de resolução é o seguinte:

abrir ambos os ficheiros; um em modo leitura (o ficheiro fonte) outro em modo escrita (o ficheiro alvo)

ler caractere por caractere o conteúdo do ficheiro fonte e escrevê-lo para o ficheiro alvo

```
let copy_file f1 f2 =  
  let c1 = open_in f1 in  
  let c2 = open_out f2 in ....
```

a representação programática de ficheiros em OCaml designa-se de **canal** (i.e. os ficheiros são canais em OCaml)

para abrir em leitura um ficheiro de nome “fich.txt” basta invocar `open_in` “fich.txt”, dizemos que abre um canal de leitura (`channel_in`)

`open_out` funciona da mesma forma, mas para o modo escrita (canal de escrita, `channel_out`)

```
# open_in;;  
- : string -> in_channel = <fun>  
# open_out;;  
- : string -> out_channel = <fun>
```

```
...  
  try  
    while true do output_char c2 (input_char c1) done  
  with End_of_file ->  
    close_in c1; close_out c2  
  
let () = copy_file Sys.argv.(1) Sys.argv.(2)
```

um caractere é lido de um canal de leitura com recurso à função `input_char`

um caractere é escrito para um canal de output com recurso à função `output_char`

o ciclo `while true do...` garante este processo até não haver mais caracteres por tratar

não é um ciclo infinito, visto sabermos que os ficheiros, por maiores que sejam, têm um tamanho finito

(este estilo de programação com ciclos `while true` é classico em desenvolvimento de sistemas reactivos ou interactivos)

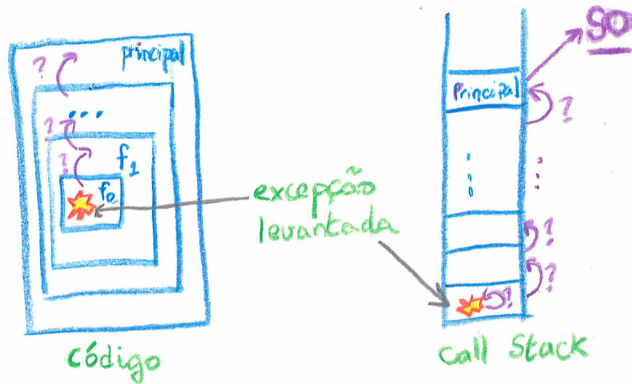
```
...
try
  while true do output_char c2 (input_char c1) done
with End_of_file ->
  close_in c1; close_out c2

let () = copy_file Sys.argv.(1) Sys.argv.(2)
```

quando a leitura esgotou o ficheiro, esta reconhece uma situação anómala à sua execução e **levanta uma excepção**: a excepção `End_of_file`

a execução é interrompida e cada bloco de expressão em execução e circundante é auscultado: este responsabiliza-se ou não pelo tratamento da excepção

a execução recomeça no primeiro bloco de expressão que, na ordem de aninhamento, trata da excepção  
no ponto onde esta excepção é tratada



se a exceção não é tratada, o processo de recuperação esgota a sua exploração dos blocos em execução, termina a execução e devolve à mão ao sistema operativo

```
...  
  try  
    while true do output_char c2 (input_char c1) done  
  with End_of_file ->  
    close_in c1; close_out c2  
  
let () = copy_file Sys.argv.(1) Sys.argv.(2)
```

Escolhemos aqui tratar da exceção no bloco que contém o ciclo `while true`

ou seja imediatamente junto do bloco que sabemos poder acionar esta exceção `input_char` lançará a exceção `End_of_file` (dentro do ciclo `while`) este não será capaz de tratar da exceção e deixará a execução na mão do bloco circundante : o bloco **try ... with**

esta construção tem precisamente por função a recuperação de exceções

aqui só pretendemos tratar da exceção `End_of_file` e caso seja essa que é apanhada neste local, o bloco prossegue com o fecho dos canais abertos (`close_in`, `close_out`)

terminamos o programa pela expressão que invoca a função principal com os argumentos provenientes da linha de comando

excepções em OCaml são... valores como quaisquer outros

em particular, podemos definir, lançar (em particular as pre-definidas), recuperar excepções

definir: (nome da excepção, começa sempre por uma maiúscula)

```
exception Nome of argumentos      (* ...of argumentos -> opcional *)
```

excepções predefinidas úteis: `End_of_file`, `Not_found`, `Invalid_argument`, `Failure`

a definição de novas excepções permite destacar situações excepcionais que pretendemos que sejam diagnosticadas e tratadas de forma destacada

lançar:

```
raise Nome          (* excepção sem argumentos*)  
raise (Nome valores) (* excepção com argumentos*)
```

funções predefinidas úteis: `(failwith msg)` e `(invalid_args msg)` (em que `msg` é a mensagem de erro que queremos associar ao evento detectado)



recuperar:

```
try
  expressão
with
  | Exn_1 -> acção_1
  | ...
  | Exn_n -> acção_n
```

semântica: avalia-se a expressão;

- se esta não lançar ou não presenciar nenhuma exceção então a execução resume-se no calculo do valor desta expressão
- se uma exceção for lançada, interrompe-se o cálculo de expressão e **filtra**-se o valor da exceção com as exceções previstas. Ao primeiro padrão `exn_i` que corresponder, retomamos a execução regular com a avaliação de `acção_i`
- se nenhum padrão corresponder, passamos a exceção ao bloco circundante

deste ponto de vista, o `try-with` tem semelhanças com uma construção condicional, assim ...

o **tipo** desta expressão é o dos diferentes resultados possíveis: todas as ramificações devem devolver valores do mesmo tipo quer seja expressão, `acção_1` ... `acção_n`!

uma **excepção** = uma **situação excepcional** durante a execução que requer tratamento excepcional (fora do decorrer normal da execução)

dualidade entre quem **detecta** uma situação excepcional  
e quem **trata** desta

a expressão que detecta uma situação excepcional é a função “*no terreno*” da execução

mas esta em geral não conhece a causa: só constata a anomalia, a consequência

quem conhece a causa é em geral a função que despoletou a avaliação/execução  
exemplo: se a função `fibonacci` tentar calcular a sequência com um argumento negativo, está sabe que há um problema, por isso responsabiliza-se por lançar a excepção correspondente

mas não sabe a origem do argumento negativo

a função que tratou de recuperar o valor para o qual se pretende a sequência, sim, tem este conhecimento: é quem deverá recuperar a excepção

## exceções como valores quaisquer

```
# exception Erro of (int*string);;
exception Erro of (int * string)
# let v = 5;;
val v : int = 5
# let e = if v > 10 then Erro (v,"mensagem de erro\n") else Erro (0,"Falha\n");;
val e : exn = Erro (0, "Falha\n")
# let f () =
  let x = read_int () in
  let y = read_int () in
  try
    let res = x / y in print_endline ("resultado = "^(string_of_int res))
  with Division_by_zero -> prerr_string "Cuidado!\n" ; raise e;;
val f : unit -> unit = <fun>
# try f () with Erro (i,msg) -> if i=0 then print_string msg;;
3
6
resultado = 0
- : unit = ()
# try f () with Erro (i,msg) -> if i=0 then print_string msg;;
4
0
Falha
- : unit = ()
```

várias funções de escrita em canais de escrita

```
output_char : out_channel -> char -> unit
output_string : out_channel -> string -> unit
output_value : out_channel -> 'a -> unit
```

destaca-se em particular a função `output_value` que *serializa* (em inglês: *Marshalling*) o seu parâmetro arbitrariamente complexo (i.e. transforma o valor num formato que possa ser escrito num ficheiro) é uma função polimórfica (ver mais adiante)

esvaziar o buffer de escrita:

```
flush : out_channel -> unit
```

várias funções de leitura de canais de leitura

```
input_char : in_channel -> char
input_line : in_channel -> string
input_value : in_channel -> 'a
```

a função `input_value` lê um valor de tipo qualquer que se encontra *serializado* no canal de leitura (e.g escrito pela função `output_value`)

a função `fprintf` (do Módulo `Printf`) também permite a escrita em canais de escrita

```
# let cout = open_out "fich.data" in
  output_value cout (1,3.14,true);
  close_out cout;;
- : unit = ()
let cin = open_in "fich.data" in
let (a,b,c) : int * float * bool = input_value cin in c;;
- bool: true
```

a serialização é um processo muito sensível: **a tipagem** (ou a execução sem interrupções acidentais/abruptas) **está vinculada à boa utilização destas funções de serialização...**

se lê algo de serializado, tem de ter a certeza que o que lá está escrito tem a forma do que quer ler!

é por isso aconselhado que indique o tipo pretendido (excepcionalmente) do valor lido por `input_value`

está a usar o **lado negro** da força! caso erra:

segmentation fault

funcionamento geral do scanf (e as suas variantes):

| scanf | entrada | formato | processamento |
|-------|---------|---------|---------------|
|-------|---------|---------|---------------|

entrada: de onde vamos retirar as leituras

formato: formato esperado das leituras

processamento: uma função que vai processar o que foi lido e extraído

- `bscanf`: entrada = um buffer de "scanning" (preferir esta função a `fscanf`)
- `sscanf`: entrada = uma string
- `scanf`: entrada é o `stdin` (por isso é omitida)
- `fscanf`: entrada = um canal de leitura

um aviso: ter cuidado com os processo de leitura que misturam as funções de tipo `read_int` com funções de tipo `scanf` (não vá o `scanf` deixar o `'\n'` ou outros caracteres para a leitura seguinte....)

## sobre o uso da função scanf

```
let f1 a b = (a,b)
let f2 a b = (a+1, b/2)
let f3 a b = a*b
let rec fact_fast x a = if x < 1 then a else fact_fast (x-1) (x*a)
let st = "ola 7 tudo bem 8"
```

```
(** ler dois inteiros de st e devolvê-los na forma de um par **)
# let v1 = sscanf st "ola %d tudo bem %d" f1;;
val v1 : int * int = (7, 8)
(** ler dois inteiros "a" e "b" de st e devolver o par (a+1,b/2) **)
# let v2 = sscanf st "ola %d tudo bem %d" f2;;
val v2 : int * int = (8, 4)
(** ler dois inteiros "a" e "b" de st e devolver a*b **)
# let v3 = sscanf st "ola %d tudo bem %d" f3;;
val v3 : int = 56
# let x = sscanf "ola 5 tudo" "ola %d tudo" (fun a -> fact_fast a 1);;
val x : int = 120
```

uso clássico de bscanf (alternativa ao scanf *por considerar*)

```
let fich = open_in "fich.txt" in
let sb = Scanf.Scanning.from_channel fich in
let x = Scanf.bscanf sb "%d" (fun a -> a) in ...
```

fora o uso de alternativas como o `bscanf`, quando é necessário misturar funções de leitura de tipos `scanf` com funções de tipo `read` **aconselha fortemente** o padrão programático seguinte que usa o `sscanf` combinado com `read_line`

```
...  
  let st = readline () in  
  let (a,b) = sscanf st " %d  %f" (fun x y -> (x,y)) in  
  let c = read_int in  
...
```

porquê? porque as funções de tipos `scanf` são predatórias e apropriam-se do canal de leitura em causa

logo, para as expressões de leitura `read` que seguem, não lhes restam aparentemente mais nada para consumir

em particular o canal para todos os efeitos atingiu o EOF

**esta situação é um bug clássico! tenham cautela!**



---

## Inverter as linhas de um texto

noções por introduzir neste exemplo

- listas
- o filtro `match-with`

o objectivo deste exemplo é ler linhas de texto da entrada standard e mostrá-las na ordem contrária (da última à primeira)

para poder realizar tal operação, vamos ler e arquivar as linhas todas e só depois proceder a visualização

basta, para arquivar as diferentes linhas, recorrer a uma estrutura de dados contentora linear e sequencial: escolhemos aqui as listas

são, na linguagem C por exemplo, as listas ligadas

OCaml disponibiliza um tipo de dado predefinido: **'a list**

que se lê: tipo das listas cujos elementos são de tipo qualquer

o tipo **'a** significa: **incógnita de tipo** ou ainda **tipo por instanciar**

```
let lines = ref []

let () =
  try
    while true do lines := read_line () :: !lines done
  with End_of_file ->
    ()

let rec print l =
  match l with
  | []      -> ()
  | s :: r -> print_endline s; print r

let () = print !lines
```

as listas (ligadas) são oferecidas em OCaml de forma primitiva  
os seus elementos têm duas formas possíveis:

a lista vazia (notação `[]`)

e a lista construída a partir de um elemento (a cabeça) e de uma outra lista (a cauda) (notação `e::l` em que `e` é o elemento em cabeça e `l` a lista na cauda)

temos aqui um exemplo de tipo que é definido de forma recursiva: uma lista é definida a partir de outra, no caso de não ser vazia

```
type 'a list = []  
             | :: of 'a * 'a list
```

as listas são **imutáveis**, uma vez construídas, não se podem alterar

a imutabilidade manifesta-se também nas funções de manipulação de listas: não alteram a lista parâmetro, devolvem uma nova lista em resultado

por isso, desde que uma lista contenha um elemento de um determinado tipo (digamos `X`) então todos os outros elementos são de tipo `X` e a lista não pode ser de outro tipo que `X list`

## as listas e o tipo dos seus elementos

```
type 'a list = []  
          | :: of 'a * 'a list
```

nesta definição temos uma manifestação explícita do conceito de **polimorfismo**

por definição, desde que os seus elementos sejam todos do mesmo tipo — única imposição nesta definição — o tipo das listas abstrai-se do tipo dos seus elementos

dizemos que as listas são **polimórficas**

assim **'a** designa o tipo **abstrato** dos elementos

é uma **incógnita de tipo** que deverá, durante a definição de uma lista ou de uma passagem de parâmetro **tomar por valor** o tipo dos elementos da lista em causa

# listas: representação e notações



$1::(4::(6::[]))$

$1::4::6::[]$

$[1; 4; 6]$

$[]$



$1::[]$



$x$

$(x)$

$y::z::u::t::[]$

$=$   
 $[y; z; u; t]$



$x::[y; z; u; t]$



$[1; 2; 3] @ [4; 5]$





```
# (@) ;;  
- : 'a list -> 'a list -> 'a list = <fun>  
# let f = (@) [1;2];;  
val f : int list -> int list = <fun>
```

neste exemplo vemos que a função de concatenação **@** é polimórfica: concatena duas listas quaisquer, desde que os seus elementos sejam do mesmo tipo (a incógnita de tipo **'a** é partilhada)

na definição de *f*, aplicamos parcialmente a função **@** dando-lhe um dos seus dois parâmetros, uma lista de inteiros

neste caso, sabemos que **@** aguarda para o seu primeiro argumento uma lista de elementos de tipo incógnita **'a**, e é-lhe fornecido uma lista de inteiros

ou seja sabemos instanciar a incógnita de tipo **'a** por **int**

como, em **@**, esta incógnita de tipo é partilhada com o tipo do segundo parâmetro, sabemos que este tem de ser igualmente de tipo **int list**  
... como o resultado da função *f*

## inverter as linhas de um texto

```
let lines = ref []
```

começamos por declarar uma referência para uma lista, inicializada para apontar para uma lista vazia

vejamos os tipos:

```
# let lines = ref [];;  
val lines : '_a list ref = {contents = []}  
# let linhas = ref ["ola"];;  
val linhas : string list ref = {contents = ["ola"]}
```

realça-se que neste momento o tipo de `lines` é algo inédito: referência para uma lista de elementos de tipo ... `'_a`

esta situação é particular: o tipo dos elementos ainda não é conhecido, mas ao ser conhecido, fica definido para sempre (as listas não podem mudar o tipos dos seus elementos no decurso da sua utilização)

esta notação significa: o tipo dos elementos desta lista ainda não é conhecido, aguarda-se à primeira ocasião para fixar esta informação de forma definitiva

esta “ocasião” é quando se conhecer o primeiro elemento da lista

## inverter as linhas de um texto

```
let () =  
  try  
    while true do lines := read_line () :: !lines done  
  with End_of_file ->  
    ()
```

as linhas de texto são lidas pela função `read_line` e colocadas à cabeça da lista apontada pela referência `lines`

quando a função de leitura de string falha (tenta ler mas não há mais linhas por ler) esta lança uma exceção `End_of_file` que quebra a execução do ciclo em que se encontra

no entanto a construção `try` apanha esta exceção e neste caso termina devolvendo `unit ()`

a referência `lines` aponta neste momento para a lista das linhas lidas **na ordem contrária!**

de facto fomos inserindo à cabeça (a última lida está a cabeça da lista)

```
let rec print l =  
  match l with  
  | []      -> ()  
  | s :: r -> print_endline s; print r  
  
let () = print !lines
```

resta-nos imprimir os elementos da lista

para esse efeito definimos a função `print` cuja função é explorar a lista da esquerda para a direita e imprimir um a um os elementos desta

para tal a função é recursiva: imprimimos o elemento em cabeça da lista e em seguida vamos recursivamente imprimir os elementos da cauda

# Inverter as linhas de um texto

socorremo-nos neste caso da construção `match ... with ....`  
cuja sintaxe genérica é a seguinte:

```
match expressão with
| padrão_1 -> acção_1      (*a barra neste primeiro padrão é opcional*)
| padrão_2 -> acção_2
...
| padrão_n -> acção_n
```

é mais um operador de **filtro** (para além do `let` e do `function/fun`); mas é um operador *exclusivamente* dedicado ao filtro

descreve-se da seguinte forma: caso o **valor** da expressão tem a **forma** do `padrão_1` então devolver o valor da expressão `acção_1`  
se tiver a forma do `padrão_2` etc...

**Nota relavante:** a listagem dos padrões possíveis tem de ser **exaustiva** (**nenhum caso esquecido!**); em caso de **sobreposição** (dois ou mais padrões possíveis), o primeiro deles na ordem em que aparece será escolhido  
todas as acções devem devolver resultados do mesmo tipo: os `match` são expressões como quaisquer outras; por isso tem igualmente um tipo e todas as acções do `match` devem devolver **valores do mesmo tipo**

a construção match é muito expressiva e particularmente prática:  
podemos aninhá-las, mas os padrões também!

a título de exemplo vejamos como escrever (variações de) uma função que verifica se o seu parâmetro lista l tem *exactamente* dois elementos

```
let has_two_elements l =  
  match l with  
  | [] -> false (*|l| = 0*)  
  | e1::li -> match li with  
               [] -> false (*|l|=1*)  
               | e12 :: l1i ->  
                 match l1i with  
                   [] -> true (* |l| = 2 *)  
                   | e13 :: _ -> false (* |l| > 2*)  
  
(** mais simples ainda **)  
let has_two_elements l =  
  match l with  
  | [] -> false  
  | e11::e12::[] -> true  
  | _ -> false
```

```
(** mais simples ainda **)  
let has_two_elements l =  
  match l with  
  | [] -> false  
  | [e11;e12] -> true  
  | _ -> false
```

```
(** mais simples ainda **)  
let has_two_elements l =  
  match l with  
  | [e11;e12] -> true  
  | _ -> false
```

```
let rec leitura l =  
  try      leitura (read_line () :: l)  
  with End_of_file -> l  
  
let () = List.iter print_endline (leitura [])
```

para terminar este exemplo vamos introduzir algumas funções e factos sobre listas

```
# 1::2::3::4::[];;
- : int list = [1; 2; 3; 4]
# [1;2;3;4];;
- : int list = [1; 2; 3; 4]
# 1::'2'::3::4::[];;
  ^^^
```

Error: This expression has type char but an expression was expected of type  
int

```
# List.length [1;2;3;4;5;6;7;8;8;8;8;8;8];;
- : int = 13
# List.append [1;2;3;4;5;6] [7;8;9;10];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# [1;2;3;4;5;6]@[7;8;9;10];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# List.sort (fun a b -> compare b a) [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
- : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
# List.iter (Printf.printf - "%d -") [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
- 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - : unit = ()
```



---

## Conversões de inteiros para uma base arbitrária

## noções por introduzir neste exemplo

- iteradores e ordem superior
- polimorfismo, novamente
- função `exit`

o objectivo é converter inteiros escritos em base  $B$  ( $2 \leq B \leq 36$ ) para a base 10

a base considerada é dada pela linha de comando e os valores numéricos por traduzir são lidos da entrada standard (até encontrar o caracter de fim de ficheiro)

para cada valor numérico lido, o valor em base 10 é mostrado

```
> radix 16
A0
-> 160
AAAAA1
-> 11184801
FFFFF
-> 1048575
1F1F1F
-> 2039583
```

```
> radix 36
LOGICA
-> 1310870746
OCAML
-> 40884429
ZEBRA
-> 59454982
```

## conversões de inteiros para uma base arbitrária

```
let base = int_of_string Sys.argv.(1)

let list_of_string s =
  let digits = ref [] in
  for i = 0 to String.length s - 1 do
    digits := s.[i] :: !digits
  done;
  !digits

let digit_of_char c =
  match c with
  | '0'..'9' -> Char.code c - Char.code '0'
  | 'A'..'Z' -> 10 + Char.code c - Char.code 'A'
  | c -> Printf.eprintf "invalid character %c\n" c; exit 1
```

## conversões de inteiros para uma base arbitrária

```
let check_digit d =  
  if d < 0 || d >= base then begin  
    Printf.eprintf "invalid digit %d\n" d; exit 1  
  end  
  
let () =  
  while true do  
    let s = read_line () in  
    let cl = list_of_string s in  
    let dl = List.map digit_of_char cl in  
    List.iter check_digit dl;  
    let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in  
    Printf.printf " -> %d\n" v  
  done
```

## conversões de inteiros para uma base arbitrária

```
let base = int_of_string Sys.argv.(1)

let list_of_string s =
  let digits = ref [] in
  for i = 0 to String.length s - 1 do
    digits := s.[i] :: !digits
  done;
  !digits
```

nesta parte inicial da solução, obtemos a base do inteiro por traduzir a partir da linha de comando

como este é lido do argv, é obtido como uma sting que depois transformamos para inteiro

esta definição e inicialização da variável base pode despoletar uma exceção caso a opção na linha de comando não seja um inteiro

## conversões de inteiros para uma base arbitrária

```
let base = int_of_string Sys.argv.(1)

let list_of_string s =
  let digits = ref [] in
  for i = 0 to String.length s - 1 do
    digits := s.[i] :: !digits
  done;
  !digits
```

`list_of_string` tem por função traduzir uma string numa lista de caracteres

esta é construída com base numa referência para uma lista e num processo iterativo

varemos a string da posição 0 até ao último carácter (posição `String.length s - 1`), cada carácter lido de `s` é colocado na lista apontada por `digit` (`digits := s.[i] :: !digits`)

no fim, devolvemos a lista apontada por `digit`

notemos que este processo permite ter os dígitos de peso mais fracos nas primeiras posições da lista devolvida (o dígito mais a direita do valor numérico original está na primeira posição na lista)



## conversões de inteiros para uma base arbitrária

```
let digit_of_char c =  
  match c with  
  | '0'..'9' -> Char.code c - Char.code '0'  
  | 'A'..'Z' -> 10 + Char.code c - Char.code 'A'  
  | c -> Printf.eprintf "invalid character %c\n" c; exit 1
```

esta função trata de traduzir um dígito em base  $B$  para o seu valor em base 10

este dígito está na forma de um caracter  $c$

se este for um dígito ( $'0' \leq c \leq '9'$ ) então transforma-se  $c$  no inteiro correspondente (a diferença entre o código ascii de  $c$  com o código ascii de  $'0'$ )

se for uma caracter entre  $'A'$  e  $'Z'$  então associa-se o valor resultante da sua posição alfabética relativamente a  $'A'$ , mais 10 ( $'A'$  vale 10,  $'B'$ , 11, etc.,  $'Z'$  vale 36)

qualquer outro caracter é considerado inválido e procedemos à interrupção do programa e devolvemos a mão ao processo pai com o código de retorno 1 (que, nos Sistemas Operativos tipo Linux, significa erro): `exit 1`

```
let check_digit d =  
  if d < 0 || d >= base then begin  
    Printf.eprintf "invalid digit %d\n" d; exit 1  
  end
```

esta função utilitária testa se um dígito  $d$  está numa base compatível com os requisitos (de 0 até a  $base - 1$ )

## conversões de inteiros para uma base arbitrária

```
let () =  
  while true do  
    let s = read_line () in  
    let cl = list_of_string s in  
    let dl = List.map digit_of_char cl in  
    let () = List.iter check_digit dl in  
    let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in  
    Printf.printf " -> %d\n" v  
  done
```

a componente principal do programa é um procedimento reactivo: opera enquanto puder (até ser levantado uma excepção)  
para cada string *s* lida, calculamos a lista de caracteres *cl* que lhe corresponde  
calculamos a seguir a lista *dl* da tradução dos caracteres em valores inteiros com recurso à função `map` e à função `digit_of_char`, assim

```
map digit_of_char ['A';'B';'5'] = [digit_of_char 'A'; digit_of_char 'B';  
digit_of_char '5'] = [10;11;5]
```

## conversões de inteiros para uma base arbitrária

```
...  
  let () = List.iter check_digit dl in  
  let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in  
  Printf.printf " -> %d\n" v  
done
```

para perceber se todos os valores resultantes estão na gama certa (entre 0 e  $B - 1$ )

aplicamos a função `check_digit` sobre todos os elementos de `dl`

se algum deles não respeitar o teste da função `check_digit`, e execução é interrompida, é devolvido `()` no caso contrário

esta verificação é feita com base no iterador `iter` que aplica uma função que produz um efeito lateral sobre todos os elementos de uma lista da esquerda para a direita

`iter check_digit [10;11;5] = check_digit 10; check_digit 11; check_digit 5`

## conversões de inteiros para uma base arbitrária

```
...  
  let () = List.iter check_digit dl in  
  let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in  
  Printf.printf " -> %d\n" v  
done
```

finalmente calculamos o valor em base 10 correspondente e mostrámo-lo na saída standard

recordemos que  $dl$  é da forma  $[d_0; d_1; d_2; \dots; d_{n-1}]$  ( $d_0$  sendo o algarismo de menor peso, o mais a direita na posição original)

este cálculo é feito com base na expressão

$$\sum_{i=0}^{n-1} d_i \times base^i$$

usamos para esse efeito o método de Horner para minimizar o curso das operações por realizar no calculo do polinómio subjacente

$$d_0 + base \times (d_1 + base \times (\dots (d_{n-1} + base \times 0) \dots))$$

## conversões de inteiros para uma base arbitrária

```
...  
let () = List.iter check_digit dl in  
let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in  
Printf.printf " -> %d\n" v  
done
```

$$d_0 + base \times (d_1 + base \times (\dots (d_{n-1} + base \times 0) \dots))$$

este cálculo pode ser realizado sobre dl com recurso ao iterador `fold_right`

`fold_right f [a;b;c] init = f a (f (b (f c init)))`

como vemos o segundo argumento de `f` é um acumulador cujo valor inicial é `init` e aplicado originalmente em conjunto com o ultimo elemento da lista: o processo percorre a lista da direita para esquerda

para a função `f`, basta aqui considerar `(fun d acc -> d + base * acc)`, e o valor inicial do acumulador é 0

```

let base = int_of_string Sys.argv.(1)

let list_of_string s = (* convert é recursiva terminal *)
  let rec convert s i fin l = if i > fin then l else convert s (i+1) fin (s.[i]::l)
  in convert s 0 (String.length s - 1) []

let digit_of_char c = match c with
  | '0'..'9' -> Char.code c - Char.code '0'
  | 'A'..'Z' -> 10 + Char.code c - Char.code 'A'
  | c -> Printf.eprintf "invalid character %c\n" c; exit 1

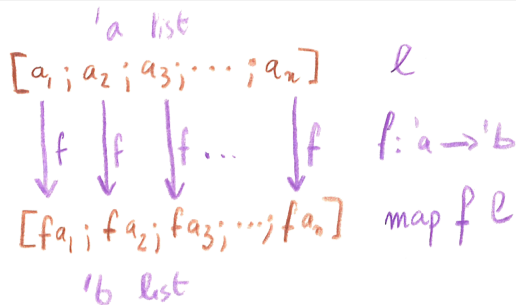
let check_digit d =
  if d < 0 || d >= base then (Printf.eprintf "invalid digit %d\n" d; exit 1)

let rec main () =
  try
    let cl = list_of_string (read_line ()) in
    let dl = List.map digit_of_char cl in
    let () = List.iter check_digit dl in
    let v = List.fold_right (fun d acc -> d + base * acc) dl 0 in
    let () = Printf.printf " -> %d\n" v in
    main ()
  with _ -> () (* em caso de erro, saímos silenciosamente*)

let _ = main ()

```

## complementos sobre listas e iteradores



```
# map (fun x -> x + 5) [1;2;3;4];; (* a lista dos valores mais 5*)
- : int list = [6; 7; 8; 9]
# map Char.code ['1'; 'a'; 'z'; 'Z'];; (* a lista dos códigos ASCII *)
- : int list = [49; 97; 122; 90]
# map Char.chr [49; 97; 122; 90];; (* a inversa *)
- : char list = ['1'; 'a'; 'z'; 'Z']
# map (fun x -> x > 10) [1;2;34;7;21];; (* a lista dos booleanos que
correspondem ao teste *)
- : bool list = [false; false; true; false; true]
```

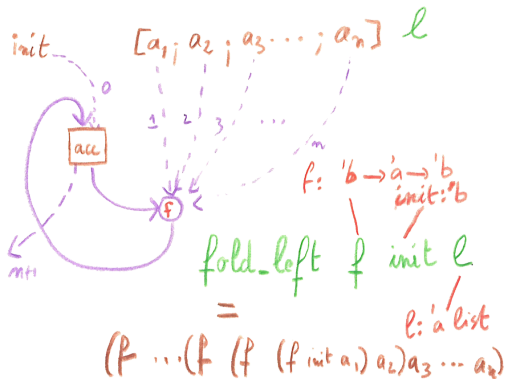


$l = [a_1; a_2; \dots; a_n]$  'a list

forall  $p$   $l = (pa_1) \wedge (pa_2) \wedge \dots \wedge (pa_n)$   
 'a  $\rightarrow$  bool 'a list

```
# for_all (fun x -> x > 10) [1;2;34;7;21];; (*todos eles? *)
- : bool = false
# exists (fun x -> x > 10) [1;2;34;7;21];; (* pelo menos um? *)
- : bool = true
# find (fun x -> x > 10) [1;2;34;7;21];; (* qual? (o primeiro)*)
- : int = 34
# partition (fun x -> x > 10) [1;2;34;7;21];; (* reparte *)
- : int list * int list = ([34; 21], [1; 2; 7])
```

# complementos sobre listas e iteradores



```
# fold_left (+) 0 [3;4;5;6;7];; (* soma dos elementos *)  
- : int = 25  
# fold_left ( * ) 1 [3;4;5;6;7];; (* produto dos elementos*)  
- : int = 2520  
# fold_left (fun a b -> a + 2*b) 0 [3;4;5;6;7];;  
- : int = 50
```

## a magia do fold\_left explicada

o fold\_left implementa este processo/algoritmo

```
let rec fold_left (f:'a->'b->'a) (v:'a) (l:'b list)=  
    match l with  
    [] -> v  
    | el::li -> fold_left f (f v el) li
```

a título de exemplo:

a própria função map pode ser escrita a custa do fold\_left e da função rev

```
let map f l = fold_left (fun a x -> f x :: a) [] (rev l)
```

e o for\_all também

```
let for_all p l = fold_left (fun b x -> b && p x) true l
```

## complementos sobre listas e iteradores

as listas associativas são uma forma cómoda de ter dicionários *on-the-fly*

a ideia é ver um par  $(x,y)$  como sendo a agregação entre chave  $x$  e conteúdo  $y$   
uma lista destes pares é assim simplesmente o dicionário com as associações até a data conhecida

juntamos informação com base na função `::` e as procuras com base na função `assoc`

```
# let l = [(1,'a'); (4,'r'); (3,'n')];; (* um dicionário predefinido*)
val l : (int * char) list = [(1, 'a'); (4, 'r'); (3, 'n')]
# List.assoc 4 l;; (* valor associado à chave 4? *)
- : char = 'r'
# List.assoc 5 l;; (*para 5?, não existe... *)
Exception: Not_found.
# let l1 = (5,'p')::l;;      (* juntar um elemento faz-se por :: *)
val l1 : (int * char) list = [(5, 'p'); (1, 'a'); (4, 'r'); (3, 'n')]
# let l2 = (1,'g')::l1;; (* juntamos mais um par *)
val l2 : (int * char) list = [(1, 'g'); (5, 'p'); (1, 'a'); (4, 'r'); (3, 'n')]
# List.assoc 5 l2;;
- : char = 'p'
# List.assoc 1 l2;; (* na procura, em caso de chaves duplicadas, é
sempre a primeira que é considerada *)
- : char = 'g'
```

---

Conclusão. Quer saber mais?

as aulas de introdução à programação OCaml apresentadas nesta UC baseam-se em duas fontes essenciais:

- **Apprendre à Programmer avec OCaml** (um *must read*!, embora em francês...).
- Sebenta **Introdução à Programação Funcional em OCaml** de Mário Pereira e Simão Melo de Sousa (link)



adicionalmente ou alternativamente, as referências seguintes introduzem OCaml de forma completa:

- **Real World OCaml**
- curso online: **Introduction to Functional Programming in OCaml** ([link](#))
- **Developing Applications with Objective Caml** ([pdf/html online aqui](#))

