Universidade da Beira Interior

Programar em OCaml
Introdução pela prática

Simão Melo de Sousa

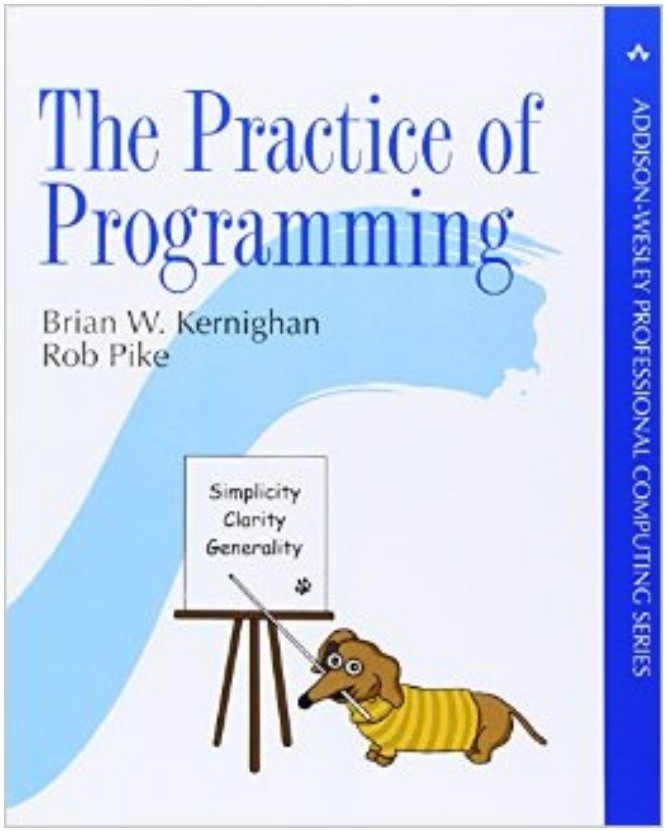
Aula 1

Programa

- Compilar, executar
- Anos bissextos
- Método de Monte-Carlo
- Desenho de uma cardióide
- Crivo de Eratósteno
- Conclusão. Quer saber mais?

Não existam boas linguagens, só existam bons programadores!

E os bons programadores **conhecem** as lições de Brian Kernighan & Robert Pike



Outro lembrete

regra de ouro:

- **cada expressão OCaml tem um valor**
- **cada expressão OCaml tem um e um só tipo**

Compilar, executar

coloquemos no ficheiro `hello.ml` o programa OCaml seguinte (uma só linha)

```
let () = print_string "Hello World!\n"
```

podemos agora compilar o programa usando um dos dois compiladores disponíveis, `ocamlc` ou `ocamlopt`

as opções em linha de comando destes compiladores são semelhantes aos do compilador `gcc` da linguagem C

```
> ocamlc hello.ml
> ./a.out
Hello World!
```

```
> ocamlopt hello.ml -o hello
> ./hello
  Hello World!
```

podemos usar igualmente o toplevel

```
>  ocaml
        OCaml version 4.02.3

# let () = print_string "Hello World!\n";;
Hello World!
# #quit;;
>
```

ou usar o tryocaml

Anos bissextos

Noções por introduzir neste exemplo

- forma geral de um programa, da sua compilação e execução
- construção `let`
- chamada de função
- tipagem e tipos de base
- sistema de módulos, directiva `open`
- bibliotecas (`Printf`, `Sys`, etc.)
- acesso aos argumentos (em linha de comando) de um programa (`Sys.argv`)
- acesso aos elementos de um vector (notação `t.(i)`)
- convenção de chamadas de função

Anos bissextos - leap_year.ml

```
let year = read_int ()
let leap =
  (year mod 4 = 0 && year mod 100 <> 0) || year mod 400 = 0
let msg = if leap then "is" else "is not"
let () = Printf.printf "%d %s a leap year\n" year msg
```

este programa é composto por 4 expressões, todas elas iniciadas pela construção `let`

exemplos de execução:

```
> ./leap_year
2016
2016 is a leap year

> ./leap_year
2015
2015 is not a leap year
```

do **processo de compilação** do programa destacamos o seguinte:

- cada expressão é analizada e compilada na ordem em que aparece
- em caso de erro, o compilador **para ao primeiro erro** encontrado
- para cada expressão é calculado **o tipo** que lhe corresponde e é comparado ao tipo opcionalmente fornecido pelo programador: é a fase de **inferência e verificação de tipos**
- um aviso. OCaml é dito **fortemente tipada**. Na prática: **OCaml é inquisitivo e sem piedade**
- porquê? **por desenho**, **transforma-se** a deteção de classes importantes de potenciais bugs em verificação de tipos

a **execução** do programa decorre da seguinte forma:

- cada expressão é calculada na ordem em que aparece
- para cada uma dela é calculada o **valor** que corresponde à expressão
- se o cálculo do valor contempla efeitos laterais, esses são realizados

a construção `let` permite a declaração de variáveis globais

qualquer declaração é necessariamente seguida da sua inicialização: **não há declaração sem inicialização**

sintaxe:

```
let variável = expressão que define o valor inicial
```

se quisermos indicar um tipo podemos realizar a declaração seguinte:

```
let (variável:tipo) = expressão que define o valor inicial
```

Anos bissextos

```
let variável = expressão que define o valor inicial
```

o processo de declaração executa-se da seguinte forma:

- **primeiro, avalia-se (calcula-se) o valor** da expressão à direira do simbolo =
- esta avaliação é acompanhada pelo cálculo do tipo da expressão
- um espaço memória para poder conter este valor é criado na memória da máquina subjacente; este espaço é calculado com base no tipo do valor
- este espaço memória passa a ser identificado pelo nome da variável; uma variável em OCaml declarada desta forma tem por **âmbito/porte** (**scope**, em inglês) o bloco de instruções que segue a sua declaração, até ao fim do ficheiro
- o valor calculado é então arquivado neste espaço
- **como na matemática**, uma variável **não mudará mais de valor** após a sua inicialização

```
let year = read_int ()
```

neste caso, a inicialização vem da chamada à função `read_int` invocada com o argumento `()`

esta função simplesmente lê da entrada standard (`stdin`) e devolve o valor inteiro lido por este meio

por isso, esta função não precisa de valor particular como argumento
assim, o valor `()` - designado de ***unit*** (que significa *"nada"*, equivalente ao `void` do C) desempenha este papel

`()` é um valor como qualquer outro, por isso tem tipo
o seu tipo é `unit`

por seu turno, o tipo `unit` é um tipo com **um só elemento**: `()`

(para efeitos de comparação, o tipo `bool`, tem dois elementos, `true` e `false`)

```
let year = read_int ()
```

se o que for lido não for um valor inteiro, um erro ocorre - mais precisamente **uma excepção é levantada** - e a execução é no presente caso interrompida

caso seja inteiro, este valor **inicializa a variável** `year`, ou seja é arquivado no espaço memória identificado pelo identificador `year`

de notar que é **inferido** o tipo `int` para a variável `year`, não foi necessário indicação do programador

Anos bissextos

```
let leap = (year mod 4 = 0 && year mod 100 <> 0) || year mod 400 = 0
```

neste caso a variável `leap` é de tipo `bool` (cujos valores são `true` ou `false`)

expliquemos em detalhe

os operadores binários `&&` e `||` são os operadores para o **e booleano (conjunção)** e o **ou booleano (disjunção)**
o operador binário inteiro `mod` calcula o **resto da divisão inteira**
o operador binário = é **a igualdade**; o operador <> é **a diferença**
(aviso: existe o == como em C, **mas o seu significado é diferente!**)

assim `leap` é `true` se o valor da variável inteira `year` for simulataneamente um múltiplo de 4 (resto da divisão por 4 é igual a 0) e não um múltiplo de 100, ou então um múltiplo de 400

`leap` é `falso` em qualquer outro caso

```
let msg = if leap then "is" else "is not"
```

a expressão condicional **if -then-else** devolve necessariamente um valor

e **tem um e um só tipo**, quer seja executado o ramo `then` quer seja executado o ramo `else`: o valor devolvido no `then` tem de ser de mesmo tipo do valor devolvido no ramo `else`

neste sentido, é equivalente ao `(teste)?bloco_1:bloco_2` da linguagem C

o ramo `else` não é opcional, **é obrigatório**

em caso de omissão, OCaml assume que o ramo `else` em falta é `else ()` (literalmente: **senão nada**) de tipo `unit` (lembram-se.... o equivalente ao `void` do C)

```
let msg = if leap then "is" else "is not"
```

assim, o valor calculado depende do valor da variável `leap`
se for true, devolve a *string* "`is`", senão devolve a "`is not`"

em ambos os ramos, os valores são de tipo **string**

após o calculo do valor inicial da variável que resulta da cálculo da condicional, o mecanismo de inferência de tipos associa à variável `msg` o tipo `string`

finalmente, a última linha

```
let () = Printf.printf "%d %s a leap year\n" year msg
```

cabe a esta expressão a impressão de uma mensagem ao utilizador conforme o ano introduzido ser um ano bissexto ou não

este objectivo é realizado pela função `printf` que está definida no módulo `Printf`
(assumimos por enquanto que *módulo* é o nome OCaml para *biblioteca*)
a sintaxe `Printf.printf` permite assim invocar a função `printf` que se encontra no módulo `Printf`

repare que a aplicação funcional em OCaml é diferente da do C, **a função e os seus argumentos são separados por espaços**

```
let () = Printf.printf "%d %s a leap year\n" year msg
```

esta função aceita em parâmetro a string `"%d %s a leap year\n"` e as duas variáveis year e msg

com base nestes dados `printf` calcula uma string (onde `%d` é substituido pelo valor de year e `%s` pelo valor de msg) e imprime o resultado no `stdout`

o calculo de `printf` não precisa de devolver um valor particular, é pouco relevante para a sua função, por isso devolve o valor `()` de tipo `unit`

é um dos exemplos em que o calculo da expressão origina efeitos laterais para além do valor final

no lugar de

```
let () = Printf.printf "%d %s a leap year\n" year msg
```

podíamos alternativamente ter usado a forma seguinte

```
let () = print_string ((string_of_int year)^''
''^msg^'' a leap year\n'')
```

onde delegamos a visualização à função `print_string` e construímos a string explicitamente com base nas variáveis `year` e `msg` e na operação de concatenação de strings `^`

a concatenação `^` concatena (valores de tipo) strings

ora, como a variável `year` não é do tipo string, **não se pode concatenar sem cuidado**

é o papel de `(string_of_int year)` que calcula a string composta dos algarismos do inteiro `year`

Anos bissextos - complementos e alternativas

o que é então a construção `let () = ...`?

é uma variante da construção `let` que se encontra nas 3 expressões iniciais do programa `leap_year.ml`

mas que aqui é usada para forçar o compilador a verificar que o valor que é devolvido pela expressão a direita do = do `let` é igual a `()`

falaremos mais adiante deste mecanismo chamado de **pattern matching**

poderíamos igualmente ter usado a construção `let _ = ...` que significa *calcular o valor, mas ignorar o resultado* (i.e. não atribuí-lo a nenhuma variável em particular)

Anos bissextos - complementos e alternativas

e se no lugar de pedir um valor do ano ao utilizador, o restirássemos da linha de comando?

para tal, basta substituir

```
  let year = read_int ()
```

por

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

```
> ocamlc leap_year2.ml -o leap_year

> ./leap_year 2013
2013 is not a leap year
```

o módulo Sys apresenta várias constantes, funções e utilitários que permitam recorrer ao sistema operativo subjacente

em particular o bem conhecido vector de string (de tipo `string array`) `argv`

funciona como em C: a posição 0 do vector `argv` contém o nome do executável, a posição 1 o primeiro argumento etc.

Anos bissextos - complementos e alternativas

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

```
> ocamlc leap_year2.ml -o leap_year
> ./leap_year 2013
2013 is not a leap year
```

aceder a uma posição particular i dum vector t faz-se pela notação $t.(i)$

assim, para aceder a posição 1 do vector `argv` que está no módulo `Sys` basta invocar `Sys.argv.(1)`

como o argumento da linha de comando é uma string, temos de usar a função de conversão `int_of_string` para `int` (que *falha* se a string não for um inteiro)

Anos bissextos - complementos e alternativas

para saber os tipos dos valores envolvidos num programa sem ser pelo toplevel

```
> ocalmc -i leap_year.ml
val year : int
val leap : bool
val msg : string
> ocamlc -i leap_year.ml > leap_year.mli
> cat leap_year.mli
val year : int
val leap : bool
val msg : string
```

os ficheiros .mli apresentam a iterface publica dos ficheiros .ml (como os ficheiros .h da linguagem C)

Anos bissextos - complementos e alternativas

algumas considerações sobre os inteiros e outros tipos numéricos, para além do tipo `int`

```
# 91;;
- : int = 91
# 0b1011011;;
- : int = 91
# 0o133;;
- : int = 91
# 0x5b;;
- : int = 91
# 2 * (7 / 2) + 7 mod 2;;
- : int = 7
# 0x2f_ff_ff_ff + 268_435_456;;
- : int = 1073741823
```

```
# max_float;;
- : float = 1.7976931348623157e+308
# min_int;;
- : int = -4611686018427387904
# 42 land (1 lsl 5);; (*5to bit de 42*)
- : int = 32
# 123l;;
- : int32 = 123l
# 10_000_000_000_000_000L;;
- : int64 = 10000000000000000L
```

numa máquina 64 bits, os valores `int` codificados **sobre 63 bits com sinal** $(-2^{62}..2^{62}-1)$, 1 bit dos 64 é usado para a gestão automatica da memória

operador de shift: `lsl`, `lsr` e `asr`

operadores bit-wise: `land`, `lor`, `lxor` e `lnot`

```
# if (if 6 > 7 then 8 = 9 else 5<3) then "ola" else "tudo bem";;
- : string = "tudo bem"
# not true && false || true;;
- : bool = true
# ((not true) && false) || true;;
- : bool = true
# 1<2 && 3 = 4 || "ola"="bomdia";;
- : bool = false
# (if 2 > 4 || true then 3 else 7) + 10;;
- : int = 13
```

os operadores booleanos tem uma ordem de avaliação particular

o operador `not` tem a prioridade mais forte

a ordem de avaliação para os operadores `&&` e `||` é da esquerda para a direita
mais, o segundo argumento de `&&` e `||` só é avaliado se for necessário

por exemplo se numa disjunção o valor do primeiro argumento é `true`, não se calcula o segundo... já sabemos que o resultado da disjunção é `true`

bool, char e string

```
# 'a';;
- : char = 'a'
# '\n';;
- : char = '\n'
# '\\';;
- : char = '\\'
# '\126';;
- : char = '~'
# '\x7E';;
- : char = '~'
# Char.code 'a';;
- : int = 97
# Char.uppercase_ascii 'a';;
- : char = 'A'
# Char.code(Char.uppercase_ascii 'a');;
- : int = 65
```

```
# "a";;
- : string = "a"
# "Hello world!\n";;
- : string = "Hello world!\n"
# ;;
- : string =
# "abc\126def";;
- : string = "abc~def"
# "abc".[1];;
- : char = 'b'
# String.length ("abc" ^ "def");;
- : int = 6
# let s = "abc";;
val s : string = "abc"
# String.uppercase_ascii s;;
- : string = "ABC"
```

```
# read_int ;;
- : unit -> int = <fun>
# read_float;;
- : unit -> float = <fun>
# read_line;;
- : unit -> string = <fun>
```

```
# print_string;;
- : string -> unit = <fun>
# print_newline;;
- : unit -> unit = <fun>
# print_endline;;
- : string -> unit = <fun>
# print_int;;
- : int -> unit = <fun>
# print_float;;
- : float -> unit = <fun>
```

dispomos igualmente das funções "a la C", como o printf e até o scanf

```
# Printf.printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
# Printf.eprintf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
# Scanf.scanf ;;
- : ('a, 'b, 'c, 'd) Scanf.scanner = <fun>
```

Abertura de módulos e o módulo Pervasive

o módulo `Pervasive` disponibiliza um conjunto importante de funções e constantes de base
por omissão, este módulo encontra-se aberto e carregado

com a excepção deste módulo, aceder aos serviços de um módulo faz-se prefixando do nome do módulo (i.e. a notação **qualificada** da função pretendida)
pode-se igualmente abrir o módulo e assim expor todos os serviços ao programa : a directiva `open`

```
# printf "ola!\n";;
Error: Unbound value printf
# Printf.printf "Olá!\n";;
Olá!
- : unit = ()
# open Printf;;
# printf "Olá!\n";;
Olá!
- : unit = ()
```

Tipagem forte

voltemos com a noção de tipagem forte de OCaml

```
# 1 + 2;;
- : int = 3
# 1 + "2";;
      ^^^
Error: This expression has type string but an expression was expected
  of type int
# 1 + 9.5;;
      ^^^
Error: This expression has type float but an expression was expected
  of type int
# (+);;
- : int -> int -> int = <fun>
```

a função OCaml + soma dois inteiros e devolve o inteiro resultado

não existe conversões implícitas realizadas pelo compilador

a conversão é sempre uma **responsabilidade do programador**, nada é feito sem a sua **atuação explícita**

o módulo `Pervasive` disponibiliza funções de conversão (`int_of_string`, etc.)

Estratégia de avaliação do OCaml

OCaml não distingue **expressões e instruções**, ambas são expressões

OCaml suporta efeitos laterais (e a programação imperativa, como veremos mais adiante)

estes estão embutidos nas expressões OCaml e são **realizados aquando da avaliação**

sem efeitos laterais a ordem de avaliação teria pouca relevância

mas na presença de efeitos laterais a ordem da avaliação **tem** ao contrário **impacto** sobre o resultado final

Estratégia de avaliação do OCaml

imagine a seguinte situação:

seja g uma função que imprime "bom" no ecrã e devolve o valor inteiro 5
seja h uma função que imprime "dia" no ecrã e devolve o valor inteiro 7

`g () + h ()` deve devolver 12 quer se calcule `g ()` antes de `h ()`
quer `h ()` antes de `g ()`

mas na presença de efeitos laterais, a ordem importa

se se avaliar da esquerda para a direita obtemos **"bomdia"** no `stdout` no caso contrário obtemos **"diabom"**

a ordem de avaliação de uma lista de expressões em OCaml **não é especificada**
no entanto, os compiladores recentes OCaml (do INRIA - que usamos aqui) escolheram avaliar da esquerda para a direita

regra de ouro:

nunca implementar programas que dependem de mecanismos com comportamento não especificado!

Estratégia de avaliação do OCaml

estudemos agora outro aspecto ligado a ordem de avaliação: **a passagem de parâmetros**

considere as seguintes expressões OCaml:

```
# let f x y = if x > 5 then x + y else x ;;
val f : int -> int -> int = <fun>
# let rec fact x = if x < 1 then 1 else x * fact (x-1);;
val fact : int -> int = <fun>
# let x = fact (4+1) + f  2 (3+4)  ;;
val x : int = 122
```

como é avaliada a última expressão? em que ordem?

OCaml segue, como a linguagem C, uma estratégia dita **ansiosa** (***eager*** em inglês):

os parametros de uma função são sempre avaliados antes da avaliação da função

Estratégia de avaliação do OCaml

```
# let f x y = if x > 5 then x + y else x ;;
val f : int -> int -> int = <fun>
# let rec fact x = if x < 1 then 1 else x * fact (x-1);;
val fact : int -> int = <fun>
# let x = fact (4+1) + f  2 (3+4)  ;;
val x : int = 122
```

olhemos para a última expressão

para avaliar, por exemplo, uma soma, primeiro avalia-se os seus dois argumentos e só depois de ter os dois valores resultantes realiza-se a soma em si

ou seja, antes de somar, temos de saber o valor de `fact (4+1)`
e o valor de `f 2 (3+4)`

como vimos, a ordem de avaliação entre estas duas expressões não é definido e depende do compilador; concretamente aqui é da esquerda para a direita

para avaliar `fact (4+1)`, **seguindo a avaliação ansiosa**, é necessário avaliar $4 + 1$ e depois entregar o valor resultante à função `fact` para sua avaliação ($5! = 120$)

Estratégia de avaliação do OCaml

```
# let f x y = if x > 5 then x + y else x ;;
val f : int -> int -> int = <fun>
# let rec fact x = if x < 1 then 1 else x * fact (x-1);;
val fact : int -> int = <fun>
# let x = fact (4+1) + f  2 (3+4)  ;;
val x : int = 122
```

para avaliar `f 2 (3+4)`, calculamos o valor de 2 e de $3 + 4$
como $2 \leq 5$, o valor de `f 2 (3+4)` é 2
finalmente 120 (i.e. 5!) $+ 2 = 122$

poder-se-á dizer que houve desperdício de cálculo, porque nem sequer y (o parametro formal que corresponde ao parâmetro efectivo $(3 + 4)$) foi necessário ao cálculo do valor de $f\ 2\ (3 + 4)$

existe uma estratégia diferente, escolhida por exemplo pela linguagem Haskell, dita de **preguiçosa** (**lazy**) que consiste em avaliar a função antes dos parâmetros estes são avaliados somente quando a própria função precisar

$(3 + 4)$ não teria assim sido calculado, neste caso

Método de Monte-Carlo

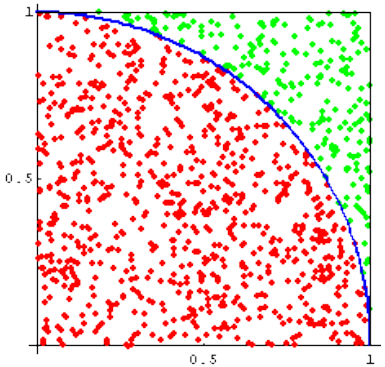
Noções por introduzir neste exemplo

- efeitos laterais
- variáveis locais (construção `let-in`)
- ciclo `for`
- números flutuantes (tipo `float`)
- módulo para a numeração aleatória `Random`

Método de Monte-Carlo

neste exemplo veremos como calcular uma aproximação (relativamente grosseira, de facto) de π

este método consiste em sortear pontos ao acaso no quadrado de coordenadas (0,0), (1,0), (1,1), (0,1) e de área 1.



a aproximação de π provém da contagem dos pontos que ficam no quarto de círculo de raio 1 e de centro $(0, 0)$

como funciona? este quarto de círculo tem por área $\frac{\pi}{4}$

assim a probabilidade de um ponto sorteado no quadrado ficar no quarto de círculo é precisamente $\frac{\pi}{4}$

em n pontos sorteados no quadrado, aproximadamente $n * \frac{\pi}{4}$ estarão dento do quarto de círculo

ou seja, π é aproximadamente $4 * \frac{n}{m}$ ($n =$ número de pontos totais sorteados, $m =$ número deste pontos que ficam no quarto de círculo)

como sabemos que um ponto está dentro do quarto de círculo sem recorrer a π? a custa da sua distância do ponto origem $(0, 0)$ que tem de ser $\leq$ a 1

Aproximação de π - monte_carlo.ml

```
let n = read_int ()

let () =
  let p = ref 0 in
  for k = 1 to n do
    let x = Random.float 1. in
    let y = Random.float 1. in
    if x *. x +. y *. y <= 1. then
      p := !p + 1
  done;
  let pi = 4. *. float !p /. float n in
  Printf.printf "%f\n" pi
```

Execução do método de Monte-Carlo

```
> ocamlopt -o monte_carlo monte_carlo.ml
> ./monte_carlo
100
3.040000
> ./monte_carlo
50000
3.149920
> ./monte_carlo
600000
3.141613
> ./monte_carlo
10000000
3.141310
> ./monte_carlo
100000000
3.141451
```

expliquemos o programa

```
let n = read_int ()
```

n é o numero de pontos por sortear
n é lido do `stdin`

o resto do programa utiliza um ciclo para ler os *n* valores

de cada vez que se entra no ciclo se o ponto sorteado estiver no quarto de círculo, incrementamos o contador de pontos incluidos no quarto de círculo

vamos estudar mais en detalhe como fazemos

primeiro

```
let () = ...
```

indica que pretendemos que a expressão que segue devolva o *unit* `()`
(indicando a intenção de efeitos laterais, por exemplo)

para poder processar o contador, temos de o declarar
pretendemos aqui declarar uma variável local ao ciclo por definir

a construção OCaml para as variáveis locais é o `let-in`

```
let ident = expressão-inicialização in expressão_onde_podemos_usar_ident
```

a variável identificada por `ident` é criada e iniializada da mesma forma do que no caso da construção `let`

a diferença fundamental prende-se com o âmbito da variável, ou seja, onde pode ser usada (onde é visível)

o **âmbito** (ou **porte**) da variável `ident` é a expressão a esquerda do `in`

termnada a execução desta expressão, **o espaço alocado à variável `ident` é liberto**, e por isso deixa de existir

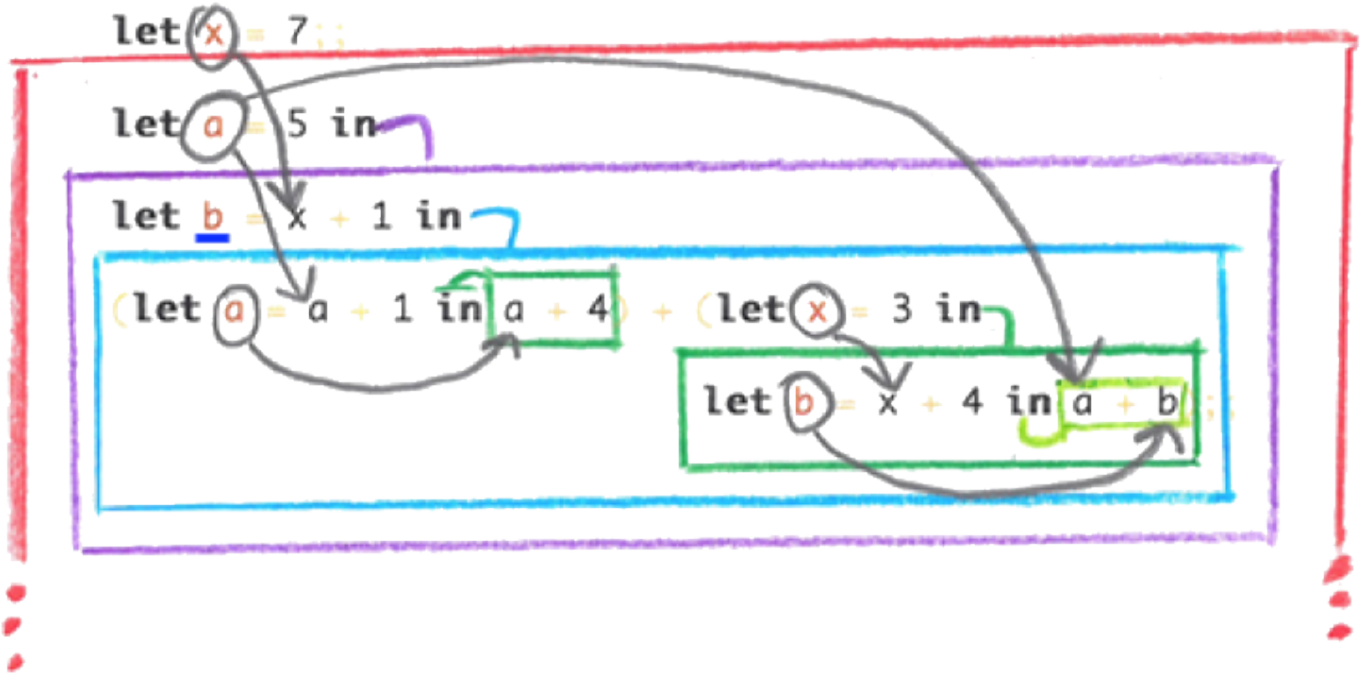
é util quando queremos controlar a porte de uma variável ou a sua utilização, ou até esconder a porte de uma outra variável

Variáveis locais

```
# let x= 10;;
val x : int = 10
# let x = 4+2 in x*x;;
- : int = 36
# x+1;;
- : int = 11
# let y = 7 in y+9;;
- : int = 16
# y+2;;
  ^
Error: Unbound value y
```

```
# let a = 1 in
  let b = 2 in
  let c = 3 in a + b + c;;
- : int = 6
# let a,b,c = 1,2,3 in a + b + c;;
- : int = 6
# (let a = 1 in a + 4) + (let b = 4 in b + 10);;
- : int = 19
```

```
# let x = 7;;
val x : int = 7
# let a = 5 in
  let b = x + 1 in
   (let a = a + 1 in a + 4) + (let x = 3 in
                               let b = x + 4 in a + b);;
Warning 26: unused variable b.
- : int = 22
```



na programação imperativa, as variáveis diferem das variáveis que encontramos na matemática: **podem mudar de valor no decorrer da sua utilização**

em programação funcional *pura*, as variáveis tem a mesma natureza que as variáveis matemáticas: **são imutáveis**

neste contexto **funcional puro**, a **mudança - o curso da computação** é confiado ao mecanismo de passagem de parâmetro

veremos este mecanismo mais em detalhe adiante

Ocaml suporta os dois tipos de variáveis, mas diferencia-os explicitamente

uma variável que muda de valor durante a *sua vida* é uma **referência**

é explicitamente referenciado como um espaço memória fixo que nunca mais muda (como uma variável matemática) mas cujo conteúdo pode ser alterado

ou seja, é uma variável padrão de OCaml mas de tipo particular

o seu acesso e a sua alteração se fazem por meios controlados

declaração

```
# let  v = ref 0;;
val v : int ref = {contents = 0}
```

acesso

```
# v;;
- : int ref = {contents = 0}
# !v;;
- : int = 0
```

alteração

```
# v := !v + 4;;
- : unit = ()
# v;;
- : int ref = {contents = 4}
# !v;;
- : int = 4
```

de notar que v não é um inteiro. É um espaço memória de tamanho suficiente para conter um inteiro: uma referência para um inteiro: **int ref**

v é uma variável tradicional OCaml: sempre será uma referência para um inteiro, e a apontar para este espaço memória particular. É **necessariamente** inicializada

o seu conteúdo pode no entanto mudar com recurso à construção **:=**, a consulta ao seu conteúdo pode ser realizado pela construção **!**

```
let p = ref 0 in ...
```

queremos criar um contador *p*, isto é uma referência para um inteiro cujo valor inteiro será incrementado
a sua definição recaí naturalmente numa referência para um inteiro, inicializado a 0

definimos assim *p* como uma variável (referência inteira) local ao procedimento de sorteio e contabilização de pontos que segue

p é inicializado pela função *ref* com o valor 0 e tem por tipo (*inferido*) **int ref**

```
for k = 1 to n do
        ...
done
```

esta parte do programa introduz um ciclo `for` controlado pelo contador k que toma valores de 1 até n, com um incremento de uma unidade

a variável k é o **índice** do ciclo e a expressão entre o `do` e o `done` é o corpo do ciclo
a execução do ciclo é resumidamente a seguinte: primeiro o índice k é inicializado (aqui para 1) e o corpo do ciclo é avaliado uma primeira vez
o índice k é incrementado (para 2) e o corpo do cilo é novamente avaliado, etc.

a execução do ciclo termina quando o corpo do ciclo é avaliado com o índice k valendo n
o **porte do índice** k é precisamente o **corpo do ciclo** (i.e. k não existe fora do corpo)

este ciclo tem por função repetir o sorteio de um ponto n vezes (de 1 a n)

```
let x = Random.float 1. in
let y = Random.float 1. in ...
```

estas duas linhas declaram **duas variáveis locais** x e y
a primeira variável tem um porte que começa no primeiro `in`
a segunda variável, declarada por um **let-in aninhado** no primeiro, tem um porte que começa no segundo `in`

a função `float` do módulo `Random` e com parâmetro z flutuante (`float`) devolve **um valor aleatório de tipo** `float` **no intervalo** $[0, z[$

pretendemos aqui usar o intervalo $[0, 1[$; para tal usamos o parâmetro flutuante 1.0 **(que podemos abreviar por** $1.$ **)**
um numéro flutuante distingue-se dum inteiro por ter um ponto na sua notação

relembre-se que em OCaml é importante distinguir e usar de acordo valores de tipos diferentes; em particular números flutuantes distinguem-se dos números inteiros

o par (x, y) é assim **um ponto ao acaso no quadrado** considerado na simulação pelo método de Monte-Carlo

```
if x *. x +. y *. y <= 1. then
    p := !p + 1
```

para saber se (x, y) está dentro do círculo basta saber se $x^2 + y^2 \leq 1$

neste caso, incrementamos p de um (encontramos mais um ponto inscrito no círculo)

mas as operações que envolvem o cálculo de $x^2 + y^2 \leq 1$ operam sobre números flutuantes e não sobre inteiros

por desenho, as operações aritméticas sobre flutuantes **distinguam-se sintaticamente** das mesmas sobre inteiros: **sufixam-se por um ponto**

mas não é o caso das comparações como $=$ ou $\leq$ que, como veremos mais adiante, são polimórficas (comparam inteiro com inteiro, flutuante com flutuante, etc. **mas não um inteiro com um flutuante!**)

notemos finalmente que usamos o operador de dereferenciação ! para aceder ao valor da referência p e que a actualizamos com o operador de atribuição :=

O tipo de um ciclo?

conforme a segunda regra de ouro de OCaml, o ciclo `for` que é uma expressão OCaml, tem de ter um tipo

mas qual é ele?

um corolário desta questão é: qual é o tipo do corpo de um ciclo?

para responder precisamos de perceber para que servem os ciclos num processo computacional, num programa

servem para **repetir** conjuntos de operações

são assim conceitos basilares da programação dita **imperativa** (em que um programa é um conjunto de ordens ao computador)

no fundo, o conceito de base que permite esclarecer esta questão é: como combinar operações (expressões, instruções, etc.)

no contexto do desenho de linguagens de programação falamos de mecanismos de **fluxo de controlo**, aqui centrados na composição de ações as linguagens de programação diponibilizam, em todas as suas variedades e paradigmas, essencialmente dois mecanismos para combinar ações:

- a composição funcional
- a sequência

Ocaml, tanto quanto a linguagem C, oferece os dois mecanismos ambos estão ao dispor, os estilos e os paradigmas de programação favorecerão mais um relativamente ao outro

admitemos a existência de duas variáveis inteiras x e y
e que pretedemos calcular

$$\frac{(x+y)^2}{2xy}$$

podemos **parcelar** o cálculo global da seguinte forma

primeiro calcular $x + y$

aplicar a seguir a função $a \mapsto a^2$ sobre o resultado

calcular $x \times y$
multiplicar o resultado por dois

dividir o resultado de $(x + y)^2$ pelo resultado de $2xy$

a **composição funcional** permite uma expressão elegante desta construção

A composição funcional

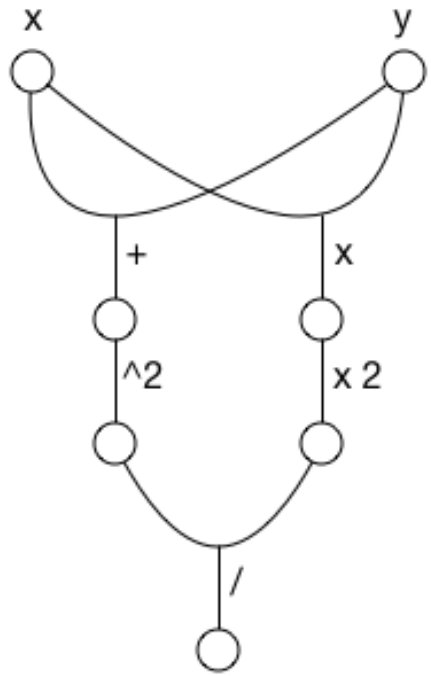
```
let square x = x * x

let result = (square (x + y)) / (2 * x * y)
```

ou então

```
let square x = x * x

let result =
    let a = x + y in
    let b = square a in
    let c = x * y in
    let d = 2 * c in
        b / d
```



A composição funcional

a composição funcional, para além de elegante, é totalmente compatível com os pressupostos do OCaml (tudo tem um valor, tudo tem um tipo)

- o calculo global é descomposto em parcelas da mesma natureza (são cálculos mais simples)
- todas as parcelas são funções: recebem parâmetros (tipados) e devolvem um valor
- o valor devolvido tem um tipo
- as entradas de umas parcelas são as saídas de outras

no exemplo anterior poderiamos ter optado por uma ordem diferente

```
let result =
   let a = y * x in
   let b = a * 2 in
   let c = y + x in
   let d = square c in
       d / b
```

porquê? **porque é aritmética e por não haver efeitos laterais envolvidos**

outra forma de compor ações é pô-las em sequência.

aqui pouco importa o resultado das ações (o valor de saída de uma ação não é necessariament relevante para uma outra ação)

as ações são vistas como caixas negras das quais só nos importa o efeito produzido

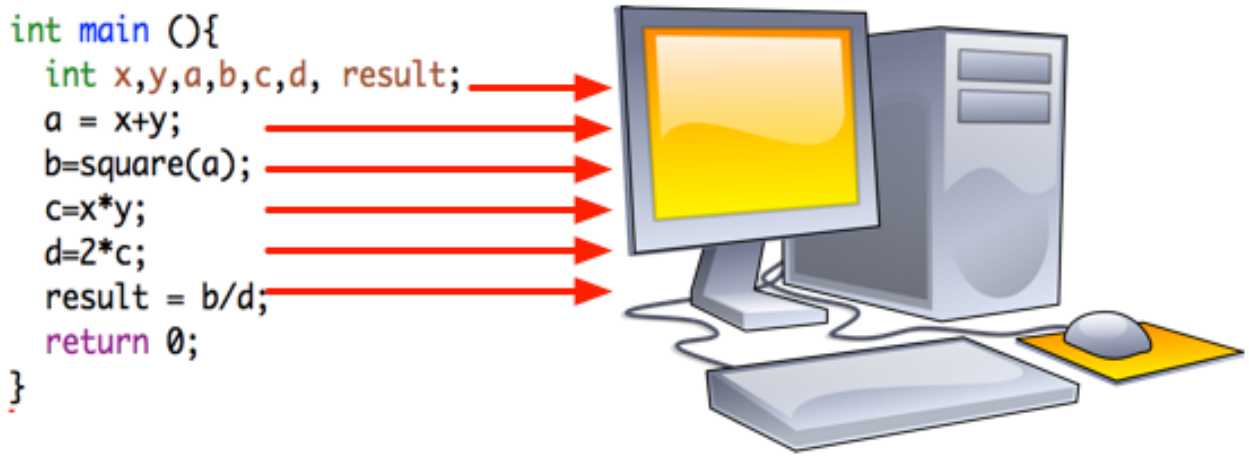
por isso interessa-nos a sequência em que estes efeitos são obtidos

```
int square (int a){return a*a;}
int main (){
  int x,y, result;
  result = square(x+y) / ( 2 * x * y);
  return 0;}
```

VS.

```
int square (int a){return a*a;}
int main (){
  int x,y,a,b,c,d, result;
  a = x+y;
  b = square(a);
  c = x*y;
  d = 2*c;
  result = b/d;
  return 0;}
```

cada instrução (com efeito lateral) refere-se e actua sobre o estado da máquina de forma implícita



por isso não precisamos de interligar as saídas das ações para as entradas de outras

em termos observacionais, cada acção da sequência faz algo –algum efeito lateral– mas **não calcula nada de particular**

O operador de sequência

na linguagem C, a sequência é dada pela listagem das ações (instruções) terminadas por um ponto-e-vírgula "`;`"

OCaml escolhe fazer como na linguagem PASCAL: a sequência de ação é dada por uma sequência de expressões **separados** por um ponto-e-vírgula **`;`**

por em sequências as expressões `A`, `B` e `C` faz-se como `A; B; C`

no contexto OCaml,

- **qual é o valor de** `A; B; C`?
- **qual é o tipo de** `A; B; C`?

vejamos...o que é relevante é o que advêm da última ação

assim, o valor desta sequência é o valor de `C`, por isso, o seu tipo é o tipo de `C`

mas, e o papel de `A` ou `B` na sequência? (...são expressões OCaml standard)

assegurar efeitos laterais (assim, o valor calculado é irrelevante): **OCaml obriga a que sejam expressões de tipo unit**

Que estilo preferir

numa linguagem como OCaml a recusividade tem um papel de destaque

obriga muitas vezes o programador a esclarecer bem as suas ideias para que tenha uma expressão elegante

mas a iteração é cómoda e até recomendada em certas situações

idealmente o programador tem de estruturar o seu programa em fases

a fase de interação com o ambiente em que o programa vai evoluir (e.g. input/output, inicialização do ambiente de trabalho etc.) costuma ser cómoda quando se usa o estilo imperativo

a fase de cálculo costuma beneficiar do estilo funcional (i.e. recursivo)

as estruturas de dados usadas também podem beneficiar um estilo relativamente ao outro
os vectores, por exemplo, comodamente podem ser explorados por um ciclo

as árvores, por seu turno, requerem naturalmente recursividade

em jeito de provocação

To iterate is human, to recurse divine — *L. Peter Deutsch*

mas não se equivocam: são estilos equivalentes

tudo o que se faz com um dos estilos pode ser feito com o outro (com mais ou menos esforço)

podemos agora voltar aos ciclos

os ciclos, como operadores de fluxo de controlo, permitam a **repetição imperativa de operações**

```
A;
for i=0 to n do
  B
done;
C
```

qual é o valor e o tipo do ciclo?
a resposta é: interessa somente o efeito logo, o valor é **()** e o tipo **unit**

qual é o valo e o tipo de B? **idem**.

os ciclos podem ser aninhados ou sequencializados, desde que seja respeitada esta política de tipos: o seu corpo produz o valor `()` e o efeito global do ciclo é igualmente o `()`

OCaml fornece a variante **downto** do ciclo `for` (decrementa o contador de 1 no lugar de o incrementar)

```
A;
for i=n downto n do
  B
done;
C
```

para estruturas cíclicas diferentes, o OCaml disponibiliza o ciclo **while**

```
A;
while (condição) do
  B
done;
C
```

De volta ao Método de Monte-Carlo

```
let n = read_int ()

let () =
  let p = ref 0 in
  for k = 1 to n do
    let x = Random.float 1. in
    let y = Random.float 1. in
    if x *. x +. y *. y <= 1. then
      p := !p + 1
  done;
  let pi = 4. *. float !p /. float n in
  Printf.printf "%f\n" pi
```

```
for k = 1 to n do
  ...
done;
```

o corpo do ciclo será avaliado n vezes, controlado pelo contador k e realiza o sorteio do n pontos no quadrado contabilizando na referência p os pontos que ficam no quarto de círculo.

tanto o corpo como o próprio ciclo devolvem `()`

```
...
  done;
  let pi = 4. *. float !p /. float n in ...
```

sucede ao ciclo o `let-in` que calcula a aproximação de π baseada no sorteio conseguido pelo ciclo

esta aproximação é feita com calculo sobre valores de tipo `float` (π não é um valor inteiro...)

mas os valores que temos para esta aproximação são inteiros (n e $!p$)
é exigida a conversão explícita: as funções `float_of_int` ou ainda `float` servem para este propósito

as operações aritméticas básicas sobre valores de tipo float são as mesmas que as da aritmética inteira, sintacticamente diferem pelo sufixo ponto (i.e. junta-se um ponto . a + para ter a soma sobre números flutuantes +.)

o módulo `Pervasive` declara mais algumas funções de base para a aritmética flutuante, o módulo `Float` igualmente

```
Printf.printf "%f\n" pi
```

finalmente procedemos a visualização da aproximação calculada

recorremo-nos da função `printf` disponível no módulo `Printf`
o seu uso é semelhante ao `printf` da linguagem C, com as habituais adaptações à sintaxe funcional OCaml (parâmetros separados por espaços)

esta expressão é a expressão final do `let () =`
é ela que devolve o resultado final da expressão envolvida
o valor de retorno de printf é ()

o que corresponde ao valor devolvido esperado
o compilador e o interpretador OCaml validam naturalmente

Complementos sobre float

```
# 5 ;;
- : int = 5
# 5. ;;
- : float = 5.
# 6e7 ;;
- : float = 60000000.
# 2 +. 3.14 ;;
Error: This expression has type int but an expression was expected of type
         float
```

```
# 3.14 +. float 2 ;;
- : float = 5.14000000000000057
# truncate 3.14159265 ;;
- : int = 3
# 4. *. atan 1. ;;
- : float = 3.1415926535897312
# -1. /. 0. ;;
- : float = neg_infinity
# 1. /. 0. ;;
- : float = infinity
```

```
# 0. /. 0. ;;
- : float = nan
# min_float ;;
- : float = 2.22507385850720138e-308
# max_float ;;
- : float = 1.79769313486231571e+308
```

Beware of the floats!

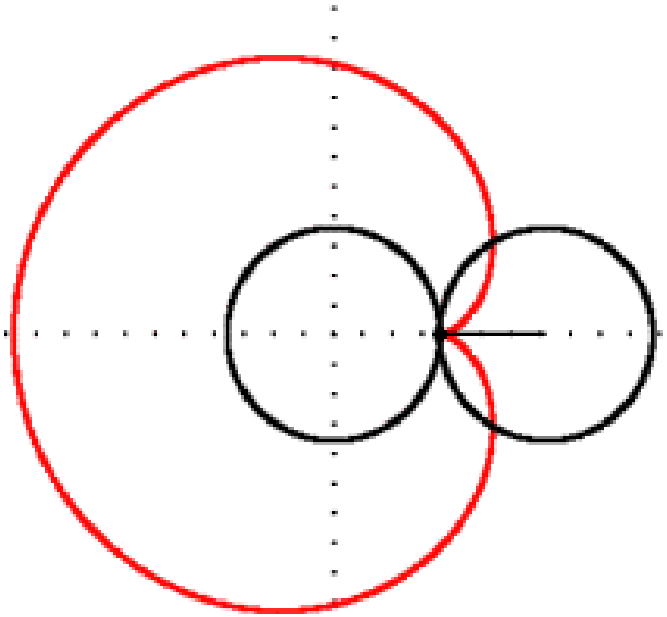
segue a norma IEEE 754 (como em C) - **são aproximações com margem de erro!**
surpresas podem advir deste facto...

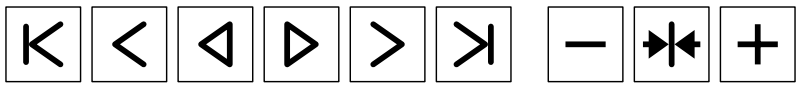
ver por exemplo:
http://floating-point-gui.de/errors/comparison/
https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

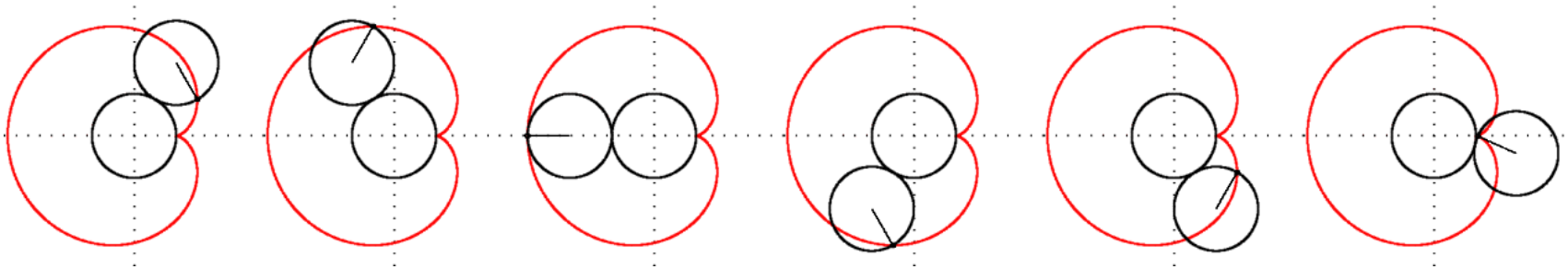
```
# let a = 0.15 +. 0.15 ;;
val a : float = 0.3
# let b = 0.1 +. 0.2 ;;
val b : float = 0.30000000000000044
# a=b;;
- : bool = false
```

Desenhar cardióides

Wikipédia: https://pt.wikipedia.org/wiki/Cardioide







Noções por introduzir neste exemplo

- bibliotecas de código objecto (extensão `.cma`)
- a biblioteca `Graphics`
- a função `ignore`

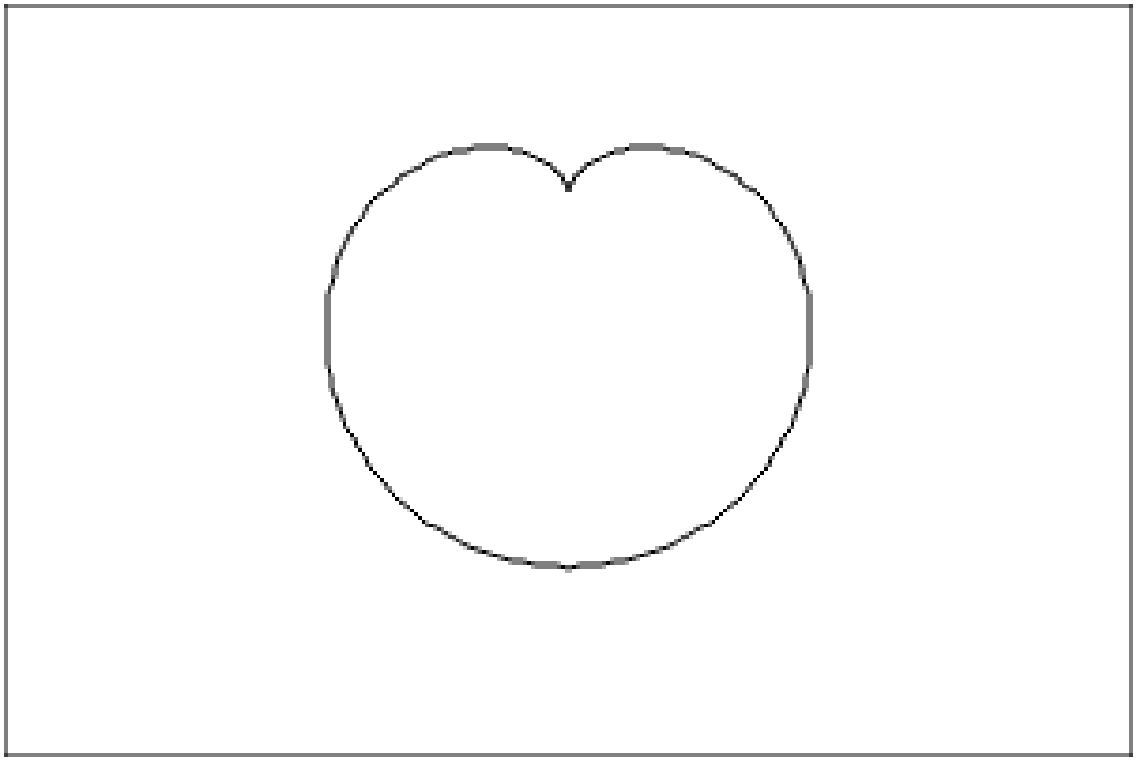
Desenhar uma cardioide

```
open Graphics

let () = open_graph " 300x200"

let () =
  moveto 200 150;
  for i = 0 to 200 do
    let th = atan 1. *. float i /. 25. in
    let r = 50. *. (1. -. sin th) in
    lineto (150 + truncate (r *. cos th))
           (150 + truncate (r *. sin th))
  done;
  ignore (read_key ())
```

Resultado da execução



do ponto de vista matemático, a cardióide que pretendemos obter (a do slide anterior) é uma curva algébrica plana que pode ser definida pela equações paramétricas seguintes

$$\begin{cases} x(\theta) &= a\,(1 - sin(\theta))\,cos(\theta) \\ y(\theta) &= a\,(1 - cos(\theta))\,sin(\theta) \end{cases}$$

onde a é o raio dos dois círculos

para desenhar a curva, vamos utilizar as funções de desenho gráfico básicas da biblioteca `Graphics`
como vamos usar várias funções da biblioteca, é cómodo evitar as chamadas qualificadas (de tipo `Graphics.f`)

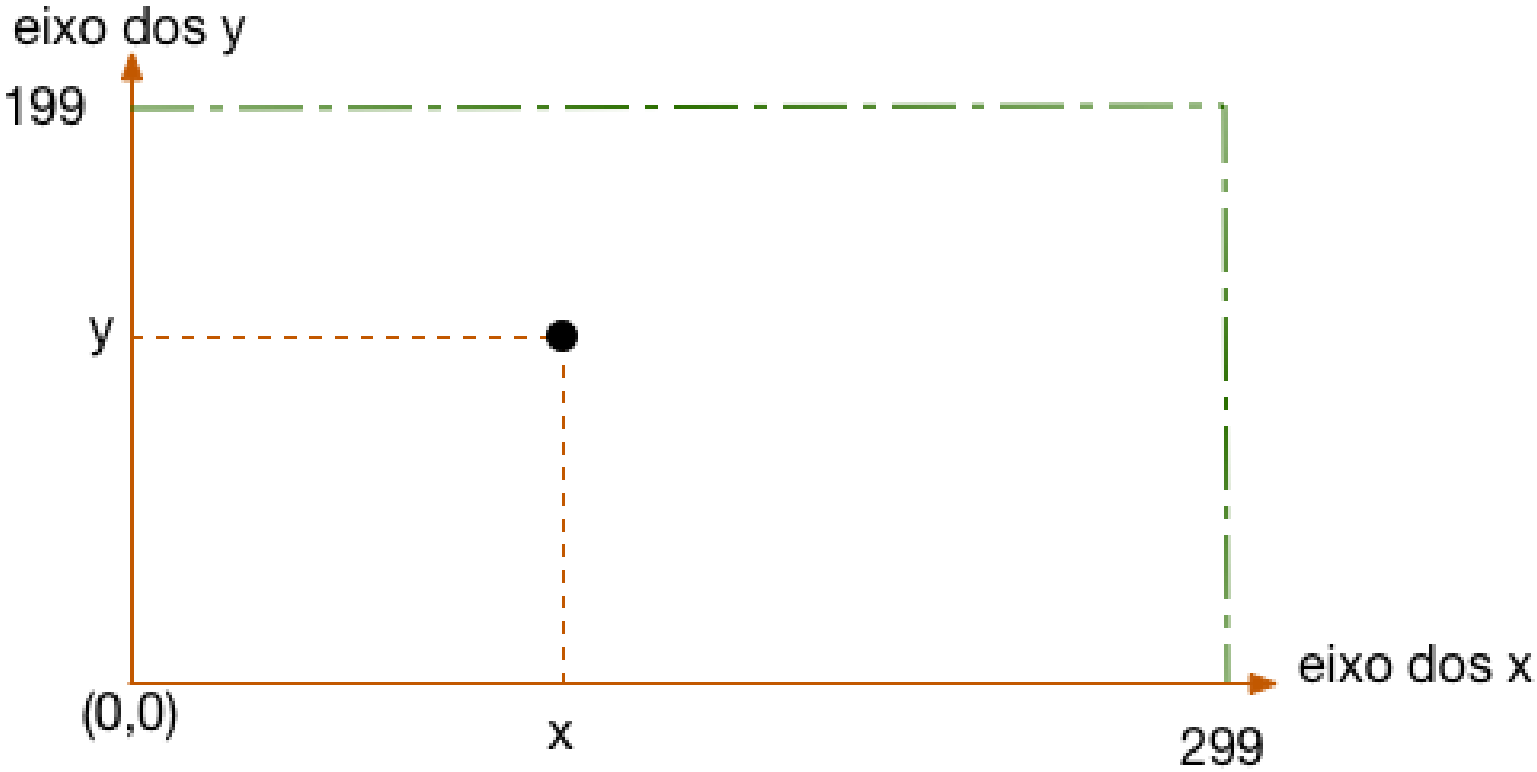
para tal vamos abrir a biblioteca, com recusto ao comando

```
open Graphics
```

```
let () = open_graph " 300x200"
```

esta expressão informa que pretendemos desenhar dentro de uma **janela** de 300 por 200 (unidade: pixel)

esta janela configura-se da seguinte forma



o resto do programa dedica-se ao desenho da curva

por isso não devolve nenhum valor em particular realizando a sua tarefa por efeito lateral (i.e. `let () =`)

as funções de desenho da biblioteca Graphics referem-se à noção de **posição corrente**

esta posição pode ser movida com a ajuda da função `moveto`

a função `lineto x y` desenha um segmente de recta do ponto corrente para o ponto (x, y) que se torna o novo ponto corrente

```
moveto 200 150
```

permite posicionar o ponto de partida do desenho da cardióde (em $(200, 150)$)

Desenhar uma cardióide

para calcular os diferentes pontos (x, y) da curva, pretendemos variar os valores de θ de 0 a 2π

para tal vamos usar um ciclo `for`

mas os ciclos for só conseguem **tratar de contadores inteiros...**

vamos tirar proveito que $arctan(1) = \frac{\pi}{4}$ assim a variação sobre θ pode ser feita a custa de um índice i tal que $\theta = arctan(1) \times \frac{i}{25}$

é desta variação sobre o i que ciclo `for` trata

para que θ varie de 0 a 2π basta que i varie de 0 a 200

```
  for i = 0 to 200 do
    let th = atan 1. *. float i /. 25. in
```

(`atan` está na biblioteca `Pervasive`, como `sin` e `cos`)

Desenhar uma cardióide

```
    let r = 50. *. (1. -. sin th) in
```

utilizamos a variável local auxiliar *r* para o cálculo de $a\ (1 - sin(\theta))$
consideramos que o raio *a* tem o tamanho de 50 píxeis

esta variável *r* permite alguma simplificação dos cálculos intermédios restantes

com

```
lineto (150 + truncate (r *. cos th))
       (150 + truncate (r *. sin th))
```

desenhamos uma linha entre o ponto corrente e $(x(\theta), y(\theta))$ (que se torna o novo ponto corrente)

como o sistema de coordenadas no `Graphics` é inteiro (a unidade é o pixel), temos de explicitamente tratar da conversão

escolhemos a função de truncatura para tal (`truncate`)

as funções de desenho aqui utilizadas realizam exclusivamente efeitos laterais, por isso devolvem todas o valor `()` de tipo `unit`

a seguir ao ciclo que desenha a curva completa, a janela gráfica fecha-se

para evitar este fenómeno, exigimos que o utilizador carregue numa tecla qualquer (função `read_key` com parâmetro (), que devolve em saída o caracter lido)

como o valor lido não tem utilidade, e que devemos devolver () (é ao que o let () = nos obriga), deitamos fora o valor

é esta a função da função `ignore` (recebe uma expressão que executa e calcula o valor, mas ignora o seu resultado, devolvendo `()`)

```
ignore (read_key ())
```

compilação com a *linkagem* à biblioteca `Graphics`

compilação para bytecode

```
> ocamlc -o cardioide graphics.cma cardioide.ml
```

ou (código nativo)

```
> ocamlopt -o cardioide graphics.cmxa cardioide.ml
```

ou ainda (para o toplevel)

```
> ocaml  graphics.cma
```

(o `ocamltry` já contempla o carregamento desta biblioteca)

Outras bibliotecas para o desenho e a interface: LalbTk, LablGtk, OCamlsdl, lablgl, etc.

Crivo de Eratósteno

Noções por introduzir neste exemplo

- vectores
- ciclo `while`
- o bloco `begin ... end`

Crivo de Eratósteno

```
open Printf

let max = read_int ()

let prime = Array.make (max + 1) true

let () =
  prime.(0) <- false;
  prime.(1) <- false;
  let limit =
     truncate (sqrt (float max)) in
  for n = 2 to limit do
    if prime.(n) then begin
      let m = ref (n * n) in
      while !m <= max do
        prime.(!m) <- false;
        m := !m + n
      done
    end
  done

let () =
 for n = 2 to max do
  if prime.(n) then printf "%d\n" n
 done
```

Crivo de Eratósteno

dado um inteiro N, pretendemos a primalidade de todos os inteiros n tais que $n \leq N$

para tal usamos um algoritmo clássico, o **crivo de Eratósteno**

este algoritmo utiliza o facto seguinte: qualquer que seja um inteiro, os múltiplos deste inteiro não são de certeza primos

em termos algoritmos basta percorrer os inteiros de forma crescente e marcar os múltiplos

imagine que estejamos a considerar o inteiro i

se este já se encontra marcado, então não é primo e de certeza que os seus múltiplos já foram marcados também, por isso a operação de marcação é excusada neste caso

se não está marcado, então é primo e é preciso marcar todos os seus múltiplos

quando parar o percurso crescente dos inteiros?

ingenuamente poderíamos simplesmente terminar este percurso em N, mas podemos fazer bem melhor

podemos parar em $\lfloor \frac{N}{2} \rfloor$ (inclusivê) porque o menor múltiplo de $\frac{N}{2}$ é precisamente $\geq N$

melhor ainda, podemos parar em $\lfloor \sqrt{N} \rfloor$ (inclusivê)

consideremos um inteiro $i > \sqrt{N}$ neste processo de marcação

qualquer múltiplo de i terá a forma $i \times k$ com $k \geq 2$

se $k < i$, então já foi eliminado (quando foi considerado k neste processo)

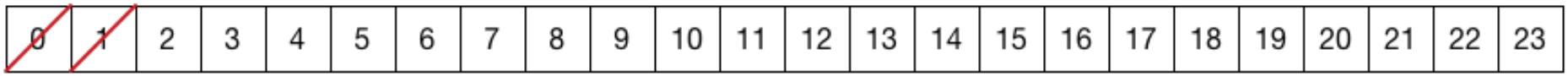
se $k \geq i$ então $i \times k > N$

vamos exemplificar o algoritmo para $N = 23$

percorremos a lista dos inteiros da esquerda para a direita até $\lfloor\sqrt{23}\rfloor$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

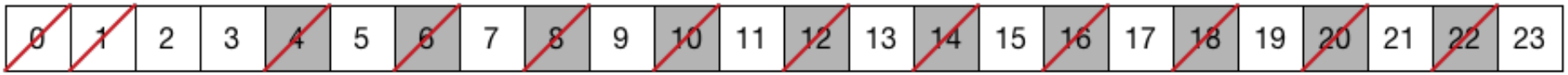
o primeiro passo é marcar 0 e 1 como não sendo primos



2 é o inteiro que segue

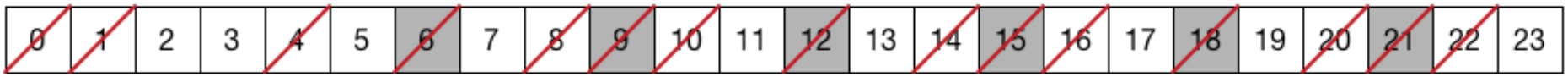
como não está marcado deixamo-lo assim (i.e. **é primo**) e marcamos todos os seus múltiplos até ao fim da sequência los até ao fim da sequência

esta marcação (células cinzentas) junta-se à marcação já realizada



3 é o inteiro que segue

de forma semelhante, como não está marcado, marcamos todos os seus múltiplos até ao fim da sequência



4 encontra-se marcado, pelo que passamos ao inteiro seguinte

5 já está para além de $\lfloor\sqrt{23}\rfloor$, o algoritmo para

os inteiros não marcados são primos.

```
let max = read_int ()
let prime = Array.make (max + 1) true
```

após a abertura do módulo `Printf`, declaramos uma variável inteira `max` inicializa pelo utilizador (com a ajuda da da função de leitura `read_int`)

esta variável tem o papel da contante *N* do exemplo anterior

declaramos e inicializamos um vector (`prime`) de comprimento $max + 1$ (de 0 até *max*) de booleanos em que cada célula é inicializada com o valor *true*
o papel da criação e da inicialização do vector cabe à função `make` (do módulo `Array`)

esta função toma em argumento o tamanho e o valor de inicialização (comum a todas às células do vector): **os vectores em OCaml tem todos um tamanho conhecido estaticamente**

como na linguagem C, os vectors tem índices inteiros e estes começam por 0
o tipo do vector é inferido pelo valor de inicialização
aqui: vector de booleanos de tamanho $max + 1$

Crivo de Eratósteno

o crivo é realizado em duas fases: a primeira calcula as marcações e a segunda imprime os inteiros que foram calculados como primos

pretendemos que o processo marque cada célula do vector `prime` com o booleano `true` se este for primo ou `false` se não for primo

a inicialização assume que todos são primos até prova do contrário

comecemos com a primeira componente

```
let () =
  prime.(0) <- false;
  prime.(1) <- false;
```

como vemos, adotamos aqui um estilo imperativo (a composição de ações é feita pelo operador de sequência), por isso indicamos que o processo devolve `()` (`let ()=...`)

para iniciar o processo de marcamos, começamos por indicar que tanto 0 como 1 não são primos
a sintaxe `t.(i) <- v` é a sintaxe que permite atribuir um valor a uma célula de um vector: a célula *i* do vector *t* recebe o valor da expressão *v*
trata-se de **acúcar sintáctico** para a função `set` da biblioteca `Array`

```
let limit =  truncate (sqrt (float max)) in
```

definimos uma variável local `limit` que toma o valor do ultimo inteiro por analizar no processo iterativo ($\lfloor\sqrt{max}\rfloor$)

notemos as conversões explícitas realizadas (por `float` e `truncate`)

a função $\sqrt{}$ é a função `sqrt` em OCaml e trabalha com flutuantes

a variável limit indica o último indice dop vector por analizar, por isso... é um inteiro

```
for n = 2 to limit do
  if prime.(n) then begin
    ...
  end
```

tendo marcado 0 e 1 podemos iterar sobre todas as células do vector de 2 até o limite (`limit`)

para tal usamos um oportuno ciclo `for` que indexa a variável *n*
se a célula de índice *n* não estiver marcada (o seu conteúdo é `true`) então marcamos os seus múltiplos

reparamos no uso de marcadores de blocos de expressões/instruções `begin ... end`

é uma comodidade sintáctica, já que as parêntesis teriam o mesmo efeito

a marcação dos múltiplos de n prossegue da seguinte forma

queremos marcar os inteiros $n \times 2$, $n \times 3$, etc... mas, se observarmos bem, se temos de marcar $n \times 2$, então este inteiro já foi marcado quando foi necessário marcar $2 \times n$ (quando marcamos os múltiplos de 2), idem para 3, ... idem até k com $k < n$

por isso basta começar a marcação em $n \times n$, ou seja basta macar os múltiplos de n a partir de n^2

mas então, que preferir? a sequência $n \times n$, $n \times (n+1)$, ... , $n \times (n+k)$, ...
ou a sequência $n \times n$, $n \times n + n$, $n \times n + n + n$, ...?

a segunda, porque só envolve uma soma a cada nova marcação (menos custosa do que uma multiplicação)

```
    let m = ref (n * n) in
    while !m <= max do
      prime.(!m) <- false;
      m := !m + n
    done
```

preferimos aqui um ciclo `while` já que o salto por realizar não é 1

a marcação terminou

assim, para terminar o programa imprimimos cada inteiro da sequência que esteja marcado pelo valor *true*

```
let () =
 for n = 2 to max do
  if prime.(n) then printf "%d\n" n
 done
```

Complementos sobre vectores e matrizes

```
let v = [| 3.14; 6.28; 9.42 |]
let v = Array.make 5  3.1

let w =
  Array.init 5 (fun i -> i * 2)
let x = 5
let v = Array.make x 3.1

v.(1)
v.(0) <- 100.0
v
v.(-1) +. 4.0
v.(30) +. 4.0
v.(4.5)
```

```
val v : float array = [|3.14; 6.28; 9.42|]
val v : float array =
    [|3.1; 3.1; 3.1; 3.1; 3.1|]

val w : int array = [|0; 2; 4; 6; 8; 10 |]
val x : int = 5
val v : float array =
    [|3.1; 3.1; 3.1; 3.1; 3.1|]
- : float = 3.14
- : unit = ()
- : float array = [|100.; 3.1; 3.1; 3.1; 3.1|]
Exception: Invalid_argument "index out of bounds".
Exception: Invalid_argument "index out of bounds".
Error: This expression has type float but
         an expression was expected of type int
```

Complementos sobre vectores e matrizes

podemos aninhar as declarações de vectores e assim construir vectores de dimensão arbitrária (mas afixada no momento da criação)

```
# let mat = [|[|1;2|];[|3;4|]|];;
val mat : int array array = [|[|1; 2|]; [|3; 4|]|]
# let mat2 =[|[|[|1; 2|]; [|3; 4|]|];[|[|1; 2|]; [|3; 4|]|]|];;
val mat2 : int array array array =
  [|[|[|1; 2|]; [|3; 4|]|]; [|[|1; 2|]; [|3; 4|]|]|]
```

ou até mesmo

```
# let t = [|   [|1|];
                  [|1; 1|];
                  [|1; 2; 1|];
                  [|1; 3; 3; 1|];
                  [|1; 4; 6; 4; 1|];
                  [|1; 5; 10; 10; 5; 1|]    |] ;;
val t : int array array =
    [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; 1|];
    [|1; 5; 10; 10; 5; 1|]|]
```

os elementos devem ser do mesmo tipo, mas se esses forem vectores, as dimensões individuais podem ser diferentes!

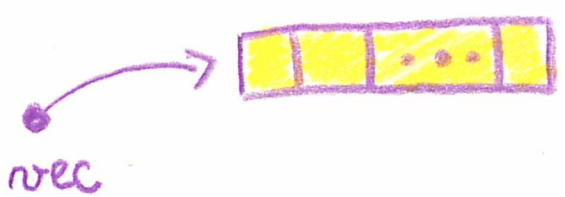
```
# mat2.(1).(0).(1);;
- : int = 2
# mat2.(1).(0).(1) <- 9;;
- : unit = ()
# mat2.(1).(0).(1);;
- : int = 9
```

os vectores são estruturas de dados imperativas (i.e. baseadas em referências)
como vimos no exemplo, podem ser alterados **in-place**, ou seja

```
let vec = [|1;2;3;...|]
```

tem por efeito na representação memória de criar uma referência de nome `vec` que aponta para um espaço de n (n = tamanho do vector) células contíguas de tamanho individual suficiente para conter um inteiro

cada uma dessas células é inicializada pelo valor calculado na inicialização do vector (a célula 0 recebe 1 etc.)



o uso de estruturas imperativas pode provocar **problemas de aliasing**, qualquer que sejam a linguagem de programação utilizada

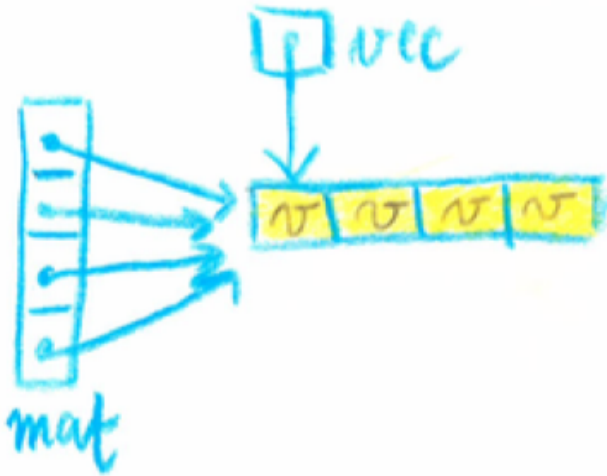
as funcionalidades imperativas de OCaml não fogem à regra

falamos de **aliasing** quando a uma determinada célula memória estão associadas mais do que uma referência (essas são ditas **alias** umas das outras)

por exemplo, imaginemos que queiramos construir uma matriz de 4 por 4 inicializada com um valor v, digamos 4.5, assim:

```
# let vec = Array.make 3 4.5;;
val vec : float array =
                [|4.5; 4.5; 4.5|]
# let mat= Array.make 4 vec;;
val mat : float array array =
  [|[|4.5; 4.5; 4.5|];
    [|4.5; 4.5; 4.5|];
    [|4.5; 4.5; 4.5|];
    [|4.5; 4.5; 4.5|]|]
```

mas a configuração memória que obtemos é deste tipo:



Manifestação indesejável do aliasing

```
# let v = Array.make 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.make 3 v;;
val m : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# m.(0).(1) <- 2;;
- : unit = ()
# m;;
- : int array array = [|[|1; 2; 0|]; [|1; 2; 0|]; [|1; 2; 0|]|]
```

Aliasing em vectores: 2 soluções padrão

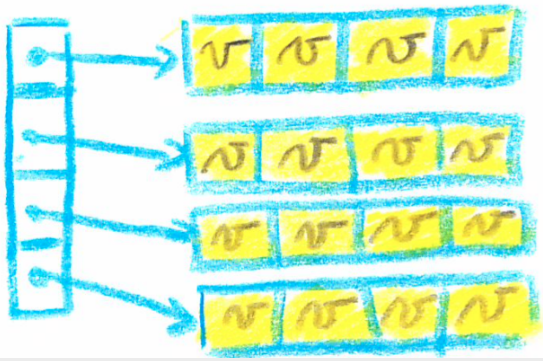
podemos clonar um vector, a função copy trata desta operação

```
# v;;
- : int array = [|1; 2; 0|]
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 2; 0|]
# v.(0) <- 40;;
- : unit = ()
# v;;
- : int array = [|40; 2; 0|]
# v2;;
- : int array = [|1; 2; 0|]
```

podemos recorrer a criação de uma matriz por uma função especializada `make_matrix` (que trata de fazer os clones necessários)

```
# let mt = Array.make_matrix 4 3 1.;;
val mt : float array array =
  [|[|1.; 1.; 1.|]; [|1.; 1.; 1.|];
    [|1.; 1.; 1.|]; [|1.; 1.; 1.|]|]
```

no caso do uso de `make_matrix`, a configuração memória é do tipo:



para terminar sobre vectores em OCaml, referimos as funções do Módulo `Array`

em particular mostremos as seguintes

```
# Array.length mt;;
- : int = 4
# Array.length mt.(0);;
- : int = 3
# let t = Array.append v v2;;
val t : int array = [|40; 2; 0; 1; 2; 0|]
# Array.sort compare t;;
- : unit = ()
# t;;
- : int array = [|0; 0; 1; 2; 2; 40|]
```

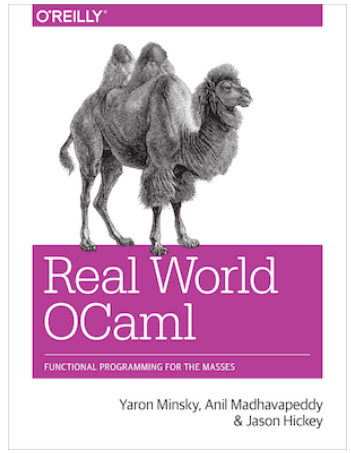
Conclusão. Quer saber mais?

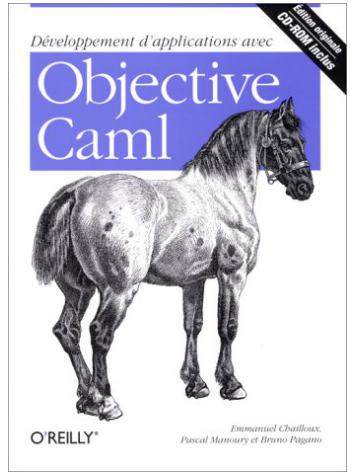
As aulas de introdução à programação OCaml apresentadas nesta UC baseam-se em duas fontes essenciais:

- **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)

Leitura aconselhada

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 onlineaqui)