

# Programação Declarativa

## Paradigmas de Programação Avançados

# Programação Funcional

*Functional  
Programming*

Modelo Funcional

# A linguagem Caml

É um ML (família de linguagens)

- Linguagem **funcional**: funções são valores de primeira classe, podem ser parâmetros
- Sintaxe simples
- Linguagem fortemente tipificada; os tipos das variáveis podem ser **inferidos**
- Suporte de padrões, filtros, polimorfismo, exceções
- Suporte para programação imperativa (!)

# O sistema OCaml

Uso do interpretador (*top level*)

Lançar o programa “ocaml” dá controle ao interpretador de comandos:

```
08:21:24 ~ khruft$ ocaml
OCaml version 4.02.3
```

```
# 2+2;;
- : int = 4
#
```

Escrevem-se expressões terminadas com “;;”

O top-level interpreta a expressão e apresenta o seu **nome**, **tipo** e o seu **valor**.

Neste caso, é *anónimo* (-), de tipo *inteiro* (int) e de valor 4.

# Definir um nome

Associar um **valor** a um **nome**.

```
# let a = 1;;  
val a : int = 1  
# let b = a+1;;  
val b : int = 2  
# a + b;;  
- : int = 3
```

O nome passa a designar o valor que lhe foi associado na instrução **let**.

# Valores

Os valores são **tipificados**. Um tipo denota um conjunto de valores, e um valor tem exatamente um tipo.

Expressão e extensão dum tipo:

- Expressão denota **como** é que se constroem valores.
- Extensão denota **todos** os valores, concretos.

Denotaremos a extensão dum tipo **t** como **ext(t)**

# Tipos de base

No Caml temos tipos fundamentais, que designamos como **tipos de base**:

- **bool** cuja extensão são as constantes **true** e **false**
- **int** cuja extensão são os inteiros num intervalo determinado pela implementação (p/ex  $\text{ext}(\text{int}) = [-2^{30} .. 2^{30}-1]$ )
- **float** que denota números em vírgula flutuante, num intervalo e com precisão determinados pela implementação
- **char** que representa os caracteres cujo código varia de 0 a 255
- **string** que denota as cadeias de caracteres, cujo comprimento máximo depende da implementação
- **unit** cuja extensão é o valor **()** que representa “nada”

# Definições ou “bindings”

Podemos definir um nome **global**:

```
# let xpto : int = 123;;  
val xpto : int = 123  
#
```

Em que estamos a introduzir um nome (xpto), para o qual indicamos o tipo (int) e um valor (123).

Também podemos definir um nome para ser usado numa expressão, como por exemplo:

```
# let x = 3 in 2*x*x - 5*x +3;;  
- : int = 6  
#
```

O nome “x” não é conhecido na instrução seguinte.



# Bindings

## Exemplo

```
# let x = 1;;  
val x : int = 1  
# x;;  
- : int = 1  
# let x = 3 in 2*x*x - 5*x +3;;  
- : int = 6  
# x;;  
- : int = 1  
# let x = 5 in x;;  
- : int = 5  
#
```

# Instrução **let**

Tem a sintaxe (EBNF):

```
let nome [ : tipo ] = expr  
{ and nome [ : tipo ] = expr }  
  [ in expr ]
```

As expressões dos valores dos nomes não podem referir os nomes em si.

# Instrução condicional

O if-then-else:

```
if  $\text{expr}_{\text{condição}}$  then  $\text{expr}_1$  else  $\text{expr}_2$ ;;
```

Tem algumas condicionantes:

- O tipo de  $\text{expr}_{\text{condição}}$  tem de ser **bool**.
- Os tipos de  $\text{expr}_1$  e  $\text{expr}_2$  têm de coincidir.

O resultado é  $\text{expr}_1$  se  $\text{expr}_{\text{condição}}$  for **true**, e  $\text{expr}_2$  caso contrário.

# Condicional - exemplos

Exemplo normal:

```
# let idade = 19;;  
val idade : int = 19  
# if idade < 18 then "não vota" else "vota";;  
- : string = "vota"
```

Exemplo redundante:

```
# let x = true;;  
val x : bool = true  
# if x = true then true else false;;  
- : bool = true  
# x;;  
- : bool = true
```

# Funções

Uma função é um valor.

Um literal de função é construído com o operador **function**, assim:

```
function arg -> expr
```

Em que **arg** é um nome que se presume ocorra em **expr**. Por exemplo:

```
# function x -> x + 1;;  
- : int -> int = <fun>
```

# Funções

Como todos os valores, podemos usá-los ou associá-los a um nome, ex:

```
# let mais_um = function x -> x + 1;;  
val mais_um : int -> int = <fun>  
# mais_um;;  
- : int -> int = <fun>  
# mais_um 23;;  
- : int = 24
```

# Funções

Também podem ser objeto de definições locais:

```
# let quad = (function x -> x*x) in quad 2;;  
- : int = 4
```

Ou ainda:

```
# let quad = (function x -> x*x) in quad (quad 2);;  
- : int = 16
```

Os parêntesis são necessários:

```
# let quad = (function x -> x*x) in quad quad 2;;
```

Characters 34-38:

```
let quad = (function x -> x*x) in quad quad 2;;  
                                ^^^^
```

Error: This function has type `int -> int`

It is applied to too many arguments; maybe you forgot a ``;'`.

# Funções

Também podemos usar funções sem ter de lhes dar nome:

```
# function x -> x * 2;;  
- : int -> int = <fun>  
# (function x -> x * 2) 3;;  
- : int = 6
```



# Tipo numa função

Pelos exemplos podemos deduzir que uma função com um argumento de tipo **A** e resultado de tipo **B** tem o tipo **A -> B**.

```
# let quad = function x -> x * x;;  
val quad : int -> int = <fun>
```

A função `quad` tem tipo **int -> int**, i.e. consome um inteiro e produz outro inteiro.

```
# let pi = 3.14;;  
val pi : float = 3.14  
# let acirc = function r -> pi *. r *. r;;  
val acirc : float -> float = <fun>  
# acirc 2.0;;  
- : float = 12.56
```

# Tipo paramétrico

Uma expressão pode admitir vários tipos, i.e. ser **polimorfica**.

Por exemplo, uma função:

```
# let foo = function x -> x :: [] ;;  
val foo : 'a -> 'a list = <fun>  
# foo 123;;  
- : int list = [123]  
#
```

Muitas funções pré-definidas são polimorficas:

# Operadores

Vimos os aritméticos, de comparação, para inteiros (+, -, \*, /, ...)

Existem operadores específicos para **float**: acrescenta-se um ponto (.) depois do operador “normal”.

```
# 1.2 +. 3.0;;  
- : float = 4.2
```

O Caml não tem conversão automática entre tipos numéricos. Há funções que o fazem explicitamente, p/ex

```
# float;;  
- : int -> float = <fun>  
# 2.3 +. float 1;;  
- : float = 3.3
```

# Conceitos de programação funcional

Ideia principal:

- Abstração
- Aplicação

A abstração **define** uma função, como uma expressão “empacotada”, sujeita a um parâmetro.

A aplicação é a **utilização** duma função, em que avaliamos a expressão da definição, depois de efetuar uma substituição do parâmetro com um valor concreto.

A aplicação resulta num **valor**, dito *resultado*.

# Cálculo Lambda

Uma abstração (também chamada “fecho” ou “closure”) é uma expressão relativamente à qual se isola um **nome**, que ocorre na expressão em causa.

Esse nome designa-se por “parâmetro”.

Em notação formal, um exemplo:

$$\lambda x. (x+1)$$

É expressão “ $x+1$ ” que fechamos sobre  $x$ , i.e. dizemos que o nome “ $x$ ” terá um valor (estará “ligado” a um valor) que só será conhecido quando **aplicarmos** a função.

# Cálculo Lambda

Se dissermos que:

$$\mathbf{inc} = \lambda x. (x+1)$$

Estamos a definir uma *função*, de nome **inc**, e que corresponde à expressão “x+1” em que “x” é o parâmetro usado na aplicação da função.

A notação para a aplicação é simples:

$$\mathbf{inc\ 123}$$

Em Caml usa-se a palavra reservada “function” para designar o “lambda” e “->” para o “.”, assim teríamos:

$$\mathbf{\#\ let\ inc = function\ x\ ->\ x+1;;}$$

# Notação funcional

O Caml permite definir funções dando-lhes logo um nome.

Consideremos uma forma de testar se um número é **par**.

```
00:06:46$ ocaml
OCaml version 4.05.0
```

```
# 3 mod 2 = 1;;
- : bool = true
# 3 mod 2 = 0;;
- : bool = false
```

Podemos definir uma função dum argumento x que verifica se x é par:

```
# function x -> x mod 2 = 0;;
- : int -> bool = <fun>
```

# Aplicação de função

Podemos aplicar esta definição a vários valores de  $x$ , para verificar que tem de facto a semântica pretendida (indicar se um número é par):

```
# (function x -> x mod 2 = 0) 2;;  
- : bool = true  
# (function x -> x mod 2 = 0) 5;;  
- : bool = false
```

Podemos associar a função a um nome, por exemplo:

```
# let par = function x -> x mod 2 = 0;;  
val par : int -> bool = <fun>
```

Que podemos agora usar:

```
# par 6;;  
- : bool = true  
# par 19;;  
- : bool = false
```



# Definição simplificada

Em vez de

```
# let impar = (function x -> not (par x));;
```

Podemos dizer:

```
# let impar x = not (par x);;  
val impar : int -> bool = <fun>  
# impar 33;;  
- : bool = true  
# impar 22;;  
- : bool = false  
# impar;;  
- : int -> bool = <fun>
```

# Aplicação

Note-se que foi necessário colocar parêntesis do volta do “par x”:

```
# let impar x = not (par x);;
```

Porque a sintaxe “**not par x**” designaria a aplicação duma função “**not**” com **dois** argumentos.... vejamos:

# Mais de um argumento

Para fazer uma função que tenha mais dum argumento, usamos repetidas vezes o operador “**function**”:

```
# let divisivel_por = function x -> function y ->
    (y mod x) = 0;;
val divisivel_por : int -> int -> bool = <fun>
# divisivel_por 2 4;;
- : bool = true
# divisivel_por 2 5;;
- : bool = false
```

# Argumentos múltiplos

Pode-se usar a notação simplificada, para definição de funções com vários argumentos:

```
# let divisivel_por x y = (y mod x) = 0;;  
val divisivel_por : int -> int -> bool = <fun>  
# divisivel_por 2 4;;  
- : bool = true  
# divisivel_por 2 5;;  
- : bool = false
```

Note-se que a função é exatamente a mesma que anteriormente (i.e. são duas funções com um argumento, sendo que uma delas retorna outra função.

# Operadores

Um operador é uma função com um ou dois argumentos e uma sintaxe particular (infixa), por exemplo

```
# 1 + 2;;  
- : int = 3
```

É o mesmo que:

```
# (+) 1 2;;  
- : int = 3
```

Ou seja, o operador + pode ser designado pela *função* (+):

```
# (+);;  
- : int -> int -> int = <fun>
```

# Aplicação parcial

Podemos fazer uma aplicação **parcial**:

```
# (+) 1;;  
- : int -> int = <fun>  
# let inc = (+) 1;;  
val inc : int -> int = <fun>
```

Da mesma maneira:

```
# let par = divisivel_por 2;;  
val par : int -> bool = <fun>  
# par 55;;  
- : bool = false  
# par 44;;  
- : bool = true
```

# Definições recursivas



Cuidado com os parêntesis...

Vamos definir a nossa conhecida função fatorial recursiva.

```
# let fact n = if n<1 then 1 else n*fact (n-1);;
```

Characters 34-38:

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

^^^^

**Error: Unbound value fact**

Problema: o nome “**fact**” não é conhecido dentro da sua própria definição...

Introduz-se uma variante do “**let**”, o “**let rec**”:

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

```
val fact : int -> int = <fun>
```

```
# fact 10;;
```

```
- : int = 3628800
```

# Definições múltiplas

O “**let**” tem variantes:

```
let [ rec ] nome [ : tipo ] = expr  
    { and nome [ : tipo ] = expr }  
    [ in expr ]
```

Notas:

- Se houver “rec” os nomes introduzidos são conhecidos nas expressões “valor”
- Se houver cláusula “in expr” os nomes só são válidos no interior desta, caso contrário são globais (i.e. existem no âmbito da própria instrução let)
- Os parâmetros “: tipo”, caso não sejam especificados, são inferidos



# Funções pré-definidas

Os operadores aritméticos, por exemplo:

```
# (+);;  
- : int -> int -> int = <fun>  
# (+.);;  
- : float -> float -> float = <fun>
```

Lógicos:

```
# (&&);;  
- : bool -> bool -> bool = <fun>
```

São na realidade funções que consomem dois argumentos, para as quais há uma sintaxe especial (sintaxe infixa).

# Tuplos

O Caml apresenta um operador de construção de valores **compostos**, ou tuplos.

```
# (1, 2);;  
- : int * int = (1, 2)  
# ("a minha idade", 23);;  
- : string * int = ("a minha idade", 23)  
# (1, 2, 10.5, (4,2));;  
- : int * int * float * (int * int) = (1, 2, 10.5, (4, 2))
```

São equivalentes a “structs” em C, mas... não há maneira de especificar um elemento concreto?

# Funções polimórficas

Uma função que retorna exatamente o seu argumento:

```
# let id = function x -> x;;  
val id : 'a -> 'a = <fun>
```


Note-se que o tipo tem **parâmetros**, aqui designados por 'a, que denotam “qualquer tipo”, i.e. qualquer tipo é aceite.

Exemplos:

```
# id 123;;  
- : int = 123  
# id "pois";;  
- : string = "pois"  
# id (1,2,3);;  
- : int * int * int = (1, 2, 3)
```

# Tuplos

Funções de acesso a tuplos:



Nota: são funções polimórficas!

```
# fst;;  
- : 'a * 'b -> 'a = <fun>  
# snd;;  
- : 'a * 'b -> 'b = <fun>  
# fst (1, 2);;  
- : int = 1  
# snd (1, 2);;  
- : int = 2  
# fst (1, 2, 3);;  
Characters 4-11:  
  fst (1,2,3);;  
    ^^^^^^^
```

Error: This expression has type 'a \* 'b \* 'c  
but an expression was expected of type 'd \* 'e

# Tuplos e let

Podemos usar tuplos em declarações:

```
# let (l,h,p) = (10,20,30);;  
val l : int = 10  
val h : int = 20  
val p : int = 30  
# let (l,h,p) = (10,20,30) in l*h*p;;  
- : int = 6000
```

# Padrões de caso

Podemos definir uma função por **casos**, i.e. dar uma definição para cada padrão de argumento.

```
# let nulo = function
  | 0 -> "nulo"
  | _ -> "nao nulo";;
val nulo : int -> string = <fun>
# nulo 10;;
- : string = "nao nulo"
# nulo 0;;
- : string = "nulo"
```

Esta é uma forma de implementar “cadeias de ifs” ou “switches”.

# Padrões de caso (funções)

Outro exemplo:

```
# let texto = function
  | 0 -> "zero"
  | 1 -> "um"
  | 2 -> "dois"
  | _ -> "outra coisa";;
val texto : int -> string = <fun>
# texto 0;;
- : string = "zero"
# texto 20;;
- : string = "outra coisa"
# texto 2;;
- : string = "dois"
```

# Definição de função por casos

Pode-se generalizar a definição de função:

```
function pad1 -> expr1 | pad2 -> expr2 | ... | padN -> exprN
```

Por exemplo:

```
# function 0 -> "z" | _ -> "nz";;  
- : int -> string = <fun>  
# (function 0 -> "z" | _ -> "nz") 0;;  
- : string = "z"  
# (function 0 -> "z" | _ -> "nz") 10;;  
- : string = "nz"
```



# Instrução “match”

Semelhante ao “switch” do C e outras linguagens:

```
match expr with p1 -> e1 | p2 -> e2 | ... | pN -> eN
```

Que na realidade é simplesmente notação simpática para:

```
(function p1 -> e1 | p2 -> e2 | ... | pN -> eN) expr
```

Por exemplo:

```
# match "noite" with "day" -> "dia" | "night" -> "noite" | _ -> "nao sei";;  
- : string = "nao sei"  
# match "night" with "day" -> "dia" | "night" -> "noite" | _ -> "nao sei";;  
- : string = "noite"  
# match "dia" with "day" -> "dia" | "night" -> "noite" | _ -> "nao sei";;  
- : string = "nao sei"  
# match "day" with "day" -> "dia" | "night" -> "noite" | _ -> "nao sei";;  
- : string = "dia"
```

# Fatorial, revista...

Podemos regressar à função fatorial:

```
# let rec fact = function
  0 -> 1
  | x -> x * fact (x-1);;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
# fact 2;;
- : int = 2
```

# Listas

O tipo lista é um tipo paramétrico (i.e. dizemos uma “lista de inteiros”, p/ex).

```
# [];;  
- : 'a list = []  
# [1];;  
- : int list = [1]  
# [1;2];;  
- : int list = [1; 2]  
# ["um"; "dois"];;  
- : string list = ["um"; "dois"]
```

Aqui vimos:

- A constante “lista vazia”, que é um valor admissível para qualquer tipo lista (**'a list**)
- Listas de inteiros e listas de strings

# Listas - construir

O operador “::” é o construtor das listas.

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

A notação “de lista” é só uma conveniência notacional, a representação interna das listas é com **::** e a lista vazia **[]**, à semelhança do que se faz no Prolog (**'.'**(**\_**,**\_**) e **[]**).

# Listas - comprimento

Vamos definir uma função com padrões:

- Um para a lista vazia
- Outro para as listas não vazias

Ficamos com este código possível:

```
# let rec length =  
  function []      -> 0  
    | _::x -> 1 + length x;;  
val length : 'a list -> int = <fun>  
# length [1;1;1];;  
- : int = 3  
# length [];;  
- : int = 0
```

# Listas - encarar como conjunto

Para ver se um elemento pertence a um conjunto, aplicando um padrão semelhante. Vamos definir uma função membro  $X$   $L$  que testa se  $X$  é um membro da lista  $L$ .

```
# let rec mem = function x -> function
  [] -> false
  | h::t -> if x=h then true else mem x t;;
val mem : 'a -> 'a list -> bool = <fun>
# mem 1 [2;3;4];;
- : bool = false
# mem 3 [2;3;4];;
- : bool = true
```

# Listas - membro

Também podemos exprimir com outra notação:

```
# let rec mem x l =  
  match l with  
    [] -> false  
  | h::t -> if x=h then true else mem x t;;  
val mem : 'a -> 'a list -> bool = <fun>  
# mem 0 [2;3;4];;  
- : bool = false  
# mem 4 [2;3;4];;  
- : bool = true
```

# Referências

A linguagem funcional Caml

<http://general.developpez.com/caml/caml-langage-fonctionnel/> (em francês)