As funções podem viajar em primeira classe

Simão Melo de Sousa

introdução

a nossa viagem de hoje leva-nos para a exploração da noção de função como um valor qualquer no cálculo realizado pelos programas que possamos escrever

em programação, esta noção de função como **cidadã de primeira classe**, ao mesmo título que os inteiros, ou quaisquer outros elementos de tipos de dados de uma linguagem de programação, designa-se de **expressão lambda** em referência ao cálculo-lambda, a *mãe de todas as linguagens de programação*

as expressões lambda, originalmente mecanismo de base das linguagens funcionais (onde nem sequer têm esse nome...) estão em todo lado!

de *hipster*, viraram *soooo mainstream*

o programador, mesmo em Java, não escapará ao paradigma funcional!

mas que o programador não se amofine, há boas razões para isso!

é por elas que linguagens como o Java adotaram estes mecanismos

para poder lidar verdadeiramente com funções como valores quaisquer, as linguagens de programação e os seus compiladores devem dispôr de alguns mecanismos que exploraremos aqui

assim o nosso comboio terá várias composições, das quais destacamos **OCaml** e **Java**, mas onde poderemos ocasionalmente pasear pelo **JavaScript** ou ainda **Python**

e a nossa viagem passará pelos apiadeiros seguintes:

- expressões lambda
- o conceito de fecho
- chamadas terminais
- programação por continuação

expressões lambda

um valor que representa uma função designa-se no paradigma funcional de **função anónima** ou ainda de valor/expressão funcional

no cálculo lambda, que lhe deu origem, são designadas de **termo lambda** ou de **expressão lambda**

é esta última designação que é utilizada em linguagens como Java, C♯ ou Javascript

o que significa ser **cidadã de primeira classe** para uma função numa linguagem de programação?

significa ter o mesmo estatuto que, por exemplo, o valor inteiro 5

- pode ser calculado, por exemplo,
  resultado de `3 + 2`
- pode intervir num cálculo, por exemplo
  `x + 5`
- pode ser passado como parâmetro a uma função, por exemplo
  `x=fact(5);`
- pode ser devolvido por uma função, por exemplo
  `int f (void) {return 5;}`

linguagens de programação que dispõem **em toda sua plenitude** de expressões lambda são ditas **linguagens de programação com ordem superior**

$\Longrightarrow$ porque dispõem de funções que podem receber e calcular outras funções

mas para uma função, o que é que a cidadania de primeira classe significa?

vamos começar pela passagem de parâmetro

significa poder passar uma função por parâmetro para outra função que poderá usá-la a sua conveniência

```
let rec filtro predicado lista =
 match lista with
 | [] -> [],[] (* nada a filtrar: por isso duas listas vazias *)
 | x::resto ->
    let (u,v) = filtro predicado resto in
      if predicado x then (x::u,v) else (u,x::v)
```

aqui a função `filtro` recebe uma função `predicado` como seu primeiro parâmetro, **como qualquer outro tipo de parâmetro...**
e, como segundo parâmetro, uma lista designada de ... `lista`

```
let rec filtro predicado lista =
 match lista with
 | [] -> [],[]
 | x::resto ->
    let (u,v) = filtro predicado resto in
      if predicado x then (x::u,v) else (u,x::v)
```

por seu turno, a função `predicado`, aceita em parâmetro valores retirados da lista em parâmetro e devolve um booleano

que faz a função `filtro`?

varre a lista de uma ponta à outra e conforme a função booleana `predicado`, coloca o valor `x` processado numa de duas listas devolvidas

```
# filtro maior_do_que_5 [1;3;2;8;4;9;6;5];;
- : int list * int list = ([8; 9; 6], [1; 3; 2; 4; 5])
```

onde a função `maior_do_que_5` é definida trivialmente como

```
  let maior_do_que_5 x = x>5
```

funções como valores devolvidos

imaginemos que temos as duas funções seguintes:

```
let maior_do_que_5 x = x>5

let maior_do_que_6 x = x>6
```

e o pequeno programa (artificial) seguinte

```
let x = read_int ()

let choose n = if n > 100 then maior_do_que_5 else maior_do_que_6

let predicado = choose x
```

o que será o valor da variável designada de `predicado`?

uma função...
....uma das duas funções booleanas `maior_do_que_5` ou `maior_do_que_6`, conforme o valor lido `x`

valor introduzido no teclado: -100

```
# filtro predicado [1;3;2;8;4;9;6;5];;
- : int list * int list =
      ([8; 9], [1; 3; 2; 4; 6; 5])
```

para poder falar das duas últimas características que fazem das funções cidadãs de primeira classe temos de introduzir a sintaxe do conceito central nesta parte da viagem: as **funções anónimas**

como 42 ou ainda *XLII* ou até mesmo 101010 são representações do conceito numérico 42 (em base decimal), que representação podemos dar a função matemática "**função sucessor**" sobre os inteiros?

na matemática, temos as notações seguintes:

$$\begin{array}{rccc} succ: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & x+1 \end{array}$$

mas convenhamos desde já que o nome é acessório, poderíamos ter usado alternativamente a definição

$$\begin{array}{rccc} s: & \mathbb{N} & \to & \mathbb{N} \\ & x & \mapsto & x+1 \end{array}$$

ou seja, na definição

$$\begin{array}{rccc} succ: & \mathbb{N} & \rightarrow & \mathbb{N} \\ & x & \mapsto & x+1 \end{array}$$

o que é realmente relevante é $x \mapsto x+1$ que captura a noção da função que soma um ao seu parâmetro inteiro
o nome é uma característica posterior (é lhe atribuída), uma comodidade

$x \mapsto x+1$ é uma **função anónima**, uma **expressão lambda**

em OCaml, o mecanismo de base para a definição de funções anónimas é a construção

```
function x -> body
```

só permite um parâmetro
pode parecer limitativo, mas não é... mas já la vamos

como definir então a função anónima *sucessor*?

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

e a função anónima "*sucessor do dobro*"?

```
function x -> 2 * x + 1
```

a questão natural que surge agora é

como usar tais funções anónimas?

esta questão tem dois pendentes: como usá-las como funções? como usá-las como valor?

como funções: ou seja, como podemos passar-lhes um parâmetro?

...como qualquer outra função

```
# (function x -> 2 * x + 1)   5 ;;
- : int = 11
# (function x -> 2 * x + 1)   7 ;;
- : int = 15
# (function x -> x + 1)   7 ;;
- : int = 8
```

como valores: de forma totalmente transparente, como qualquer outro valor...
podem por exemplo ser usados em definições

```
# let x = 5;;
val x : int = 5
# let f = (function x -> x + 1);;
val f : int -> int = <fun>
# f x;;
- : int = 6
```

combinando tudo, num exemplo

```
(* A "composição de funções"  <|   A notação ( op ) na definição *)
(* permite definir uma função binária op para ser usada de forma infixa*)
let ( <| ) f g = fun x -> f (g x)
```

```
# (function x -> x + 1) <| (function y -> 2 * y + 1);;
- : int -> int = <fun>
# ((function x -> x + 1) <| (function y -> 2 * y + 1))  5  ;;
- : int = 12
# ((function x -> x+1) <| (function y -> 2*y+1) <| (function z -> 10*z)) 5 ;;
- : int = 102
# let f =
  ((function x -> x + 1) <| (function y -> 2 * y + 1) <| (function z -> 10 * z));;
val f : int -> int = <fun>
# f 5;;
- : int = 102
```

voltando à definição de funcões anónimas,
como definir a função anónima "*soma dos seus três parâmetros*"?

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

hummm... isso merece uma pequena explicação
até porque não se assemelha à primeira vista a sua definição matemática natural

$$\begin{array}{rccc} f: & \mathbb{N}\times\mathbb{N}\times\mathbb{N} & \rightarrow & \mathbb{N} \\ & x,y,z & \mapsto & x+y+z \end{array}$$

na verdade, poderiamos ter usado a definição OCaml seguinte

```
function (x,y,z) -> x + y + z
```

esta definição é semanticamente equivalente à definição anterior, mas é operacionalmente menos interessante: não tira proveito de dois aspectos interessantes de linguagens de programação como OCaml, a **ordem superior** e a **avaliação parcial** resultante

esta definição é a de uma função que aceita **um** triplo de inteiros (composto por x, y e z) – e não três valores – e devolve a soma das três componentes do triplo

enquanto

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

é uma função que a um `x`, devolve uma função que a um `y` devolve uma função que a um `z` devolve a soma de x, y e z.... ufa...

vamos a seguir explorar o conceito de avaliação parcial (que justifica porque as funções anónimas aninhadas são interessantes)

mas antes introduzimos algum açúcar sintáctico:
o aninhamento da construção `function` é de tal forma comum que

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

pode ser substituido pelo uso da construção `fun`, açúcar sintáctico de `function` para esse efeito

```
fun x y z -> x + y + z
```

dizemos nestes dois casos que a função está na sua forma **currificada** (***curryfied***) - em referência ao lógico Haskell Curry e em contraposição à primeira versão da função

a **avaliação parcial** de uma função (anónima ou não) é a capacidade em realizar parcialmente a passagem de parâmetro

ou seja, passar por parâmetro parte e não a totalidade dos parâmetro definidos

comecemos por mostrar que a definição

```
function (x, y, z) -> x + y + z
```

não dispõe da possibilidade de avaliação parcial

```
# function  (x,y,z)  ->  x + y + z ;;
- : int * int * int -> int = <fun>
# (function  (x,y,z)  ->  x + y + z) (1,2,3);;
- : int = 6
# (function  (x,y,z)  ->  x + y + z) 1;;
Error: This expression has type int but an expression
 was expected of type     int * int * int
```

ao contrário da definição alternativa dada

avaliação parcial

```
# fun x y z  -> x + y + z ;;
- : int -> int -> int -> int = <fun>
# (fun x y z  -> x + y + z) 1 2 3 ;;
- : int = 6
# (fun x y z  -> x + y + z) 1 ;;
- : int -> int -> int = <fun>
# (fun x y z  -> x + y + z) 1 2 ;;
- : int -> int = <fun>
# let f = (fun x y z  -> x + y + z) 1 2 ;;
val f : int -> int = <fun>
# f 3 ;;
- : int = 6
# let g = (fun x y z  -> x + y + z) 1 ;;
val g : int -> int -> int = <fun>
# g 2 3;;
- : int = 6
# let h = g 2;;
val h : int -> int = <fun>
# h 3;;
- : int = 6
```

assim, em particular

```
let f = (fun x y z  -> x + y + z) 1 2
```

é **exactamente** a função

```
let f = (fun  z  -> 3 + z)
```

ou ainda

```
let f z =  3 + z
```

como funciona a avaliação parcial?

na base está o uso de uma construção programática, chamada se **fecho** (***closure***, em inglês)

este conceito é a representação em execução (*runtime*) de uma função digamos f

ou seja, que está a ser efectivamente executado no computador/memória quando se executa f (aplicado a parâmetros efectivos)

estudaremos a seguir e mais em detalhe a noção central de fecho que ultrapassa em muito a simples aplicação |A aplicação parcal

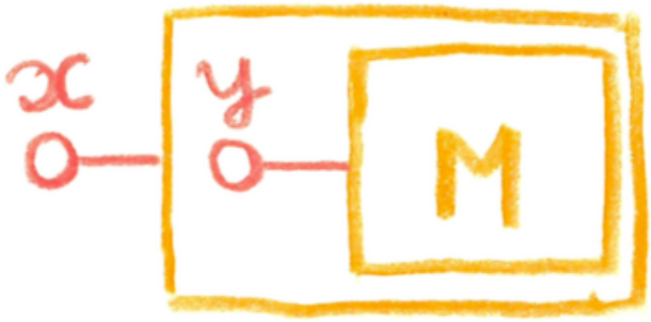
exploremos por agora somente o funcionamento da avaliação parcial

na base a função

pode ser representada graficamente por:

a função calcula M tendo em conta um valor x (aqui o parâmetro formal) que lhe será passado (o parâmetro efectivo)

no caso aninhado, temos



a *"caixa"* calculada por *function x* contém uma função sobre *y*, uma outra caixa

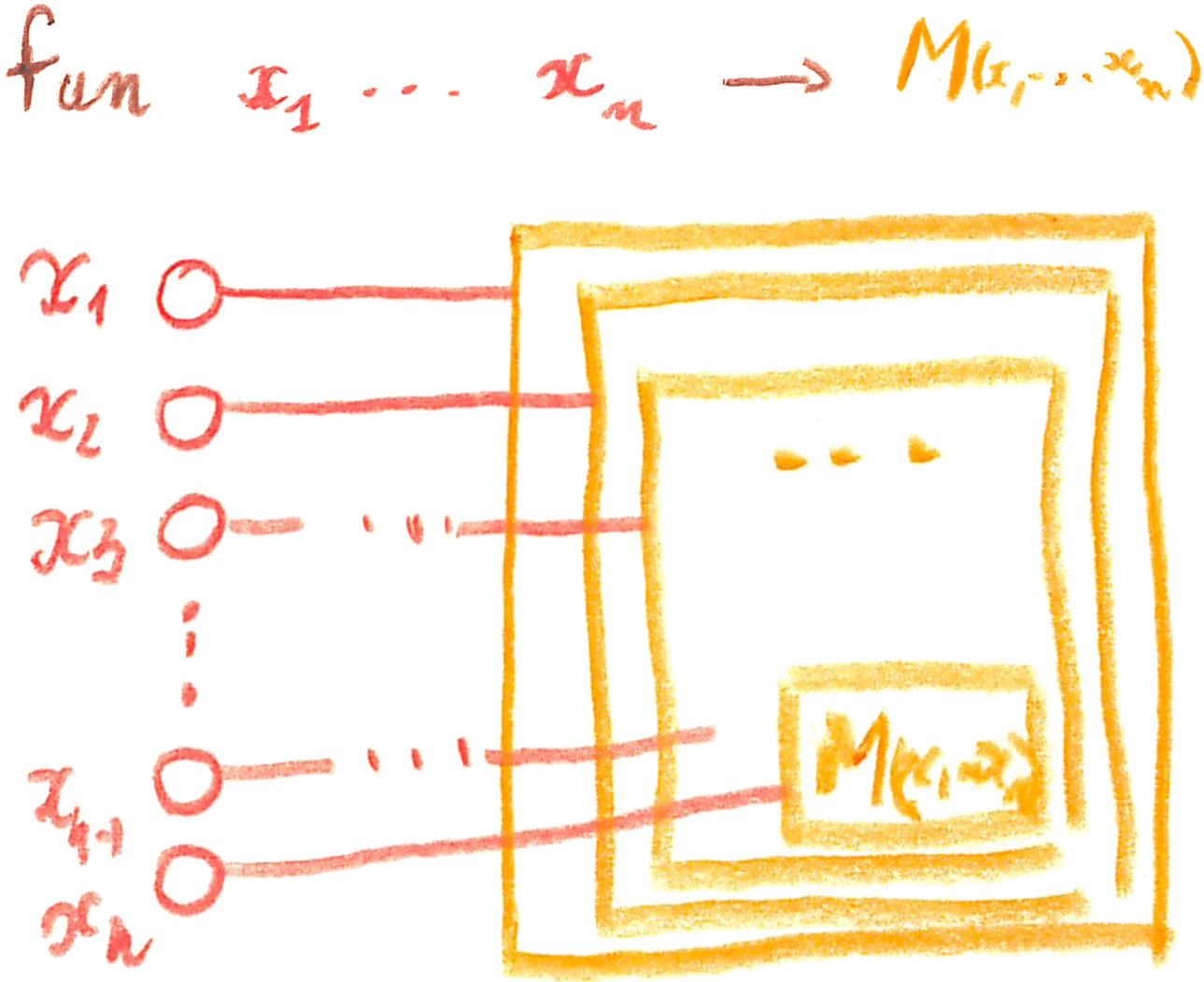
esta representação gráfica põe em evidência que conforme conveniência o aninhamento de funções pode ser visto como uma caixa com duas entradas ou uma caixa de uma entrada contendo uma caixa com uma entrada

no caso do agrupamento dos parâmetros em tuplos a situação é

percebemos graficamente porque não há aqui a alternativa de ponto de vista

ou passamos os parâmetros todos de uma vez (porque na realidade só há um...) ou nenhum

voltemos ao caso aninhado, o caso geral

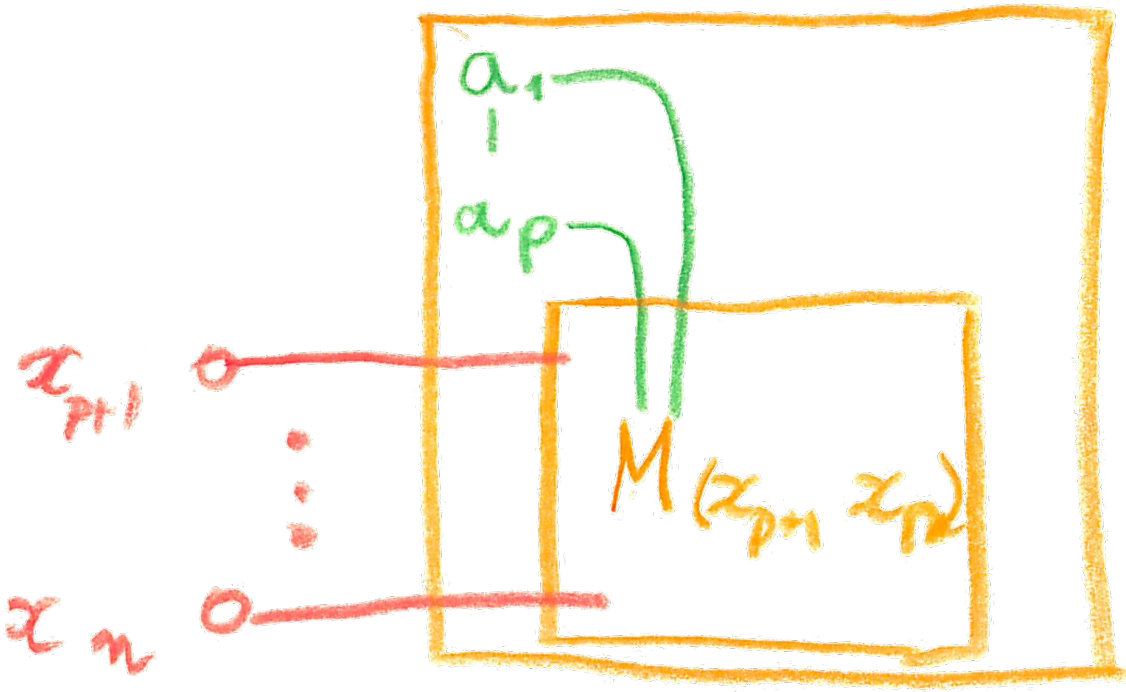


se fornecer parte dos parâmetros, digamos $a_1 \ldots a_p$

$$(\text{fun } x_1 \cdots x_n \longrightarrow M(x_1, \ldots x_n)) \quad a_1 \ldots a_p$$

$$1 \leq p \leq n$$

então a situação graficamente a seguinte



em java a sintaxe para uma **lambda expression** é:

```
  x -> M
```

por exemplo

```
import java.util.function.*;
...
  IntBinaryOperator simpleAdd = (a, b) -> a + b;
  IntFunction <IntUnaryOperator> curriedAdd = a -> b -> a + b;
...
```

onde introduzimos duas versões da mesma função anónima (a soma de dois inteiros)
a segunda versão é a versão currificada da primeira

como Java não tem inferência de tipos, temos de introduzir o tipo esperado para a operação

o package `java.util.function`, em parte listado no acetato seguinte fornece algumas das opções mais usadas

em particular

```
import java.util.function.*;
...
  IntBinaryOperator simpleAdd = (a, b) -> a + b;
  IntFunction <IntUnaryOperator> curriedAdd = a -> b -> a + b;
...
```

a variável `simpleAdd` recebe como valor uma expressão lambda que codifica uma função binária que recebe dois inteiros e que os soma

a variável `curriedAdd` é declarada como uma função unária que recebe um inteiro (o parâmetro `a`) e devolve operador unário inteiro (de tipo `IntUnaryOperator`) `b -> a + b`

como é óbvio aqui, esta segunda função é a versão currificada da soma de inteiros

Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two double-valued operands and producing a double-valued result.
DoubleConsumer	Represents an operation that accepts a single double-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one double-valued argument.
DoubleSupplier	Represents a supplier of double-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single double-valued operand that produces a double-valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation upon two int-valued operands and producing an int-valued result.
IntConsumer	Represents an operation that accepts a single int-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.

retomemos o exemplo Java

```
out.println(simpleAdd.applyAsInt(4, 5));
```

`public interface IntBinaryOperator`

applyAsInt

```
int applyAsInt(int left,
               int right)
```

Applies this operator to the given operands.

Parameters:

`left` - the first operand

`right` - the second operand

Returns:

the operator result

```
out.println(curriedAdd.apply(4).applyAsInt(5));
```

public interface
IntFunction<R>

apply

`R apply(int value)`

Applies this function to the given argument.

Parameters:

`value - the function argument`

Returns:

`the function result`

public interface
IntUnaryOperator

applyAsInt

`int applyAsInt(int operand)`

Applies this operator to the given operand.

Parameters:

`operand - the operand`

Returns:

`the operator result`

```
IntUnaryOperator junta10 = curriedAdd.apply(10);

out.println(junta10.applyAsInt(6));
```

`junta10` é uma função que resulta da aplicação parcial de `curriedAdd`

em métodos em que se esperam parâmatros de tipo *função* (como as que existam em `java.utils.functions`, por exemplo `IntUnaryOperator`)

podemos simplesmente passar uma expressão lambda, ou o seu nome (se este foi atribuído a uma variável, como a variável `junta10`)

é também possível passar um método existente, caso queiramos

nesta caso para referir-se ao seu nome basta usar a sintaxe `classe::nome`

este identificador representa o método `nome` da classe `classe`

exemplos de uso da avaliação parcial

em linguagens de programação que suportam o conceito *das funções como cidadãs de primeira classe*, a técnica da avaliação parcial oferece mecanismos cómodos de especialização de código

imaginemos que se defina uma função f cujo funcionamento dependa de vários parâmetros $x_1 \ldots x_n$

quem desenvolve este serviço só tem de se preocupar em desenvolver da forma mais geral e abrangente possível, colocando na forma de parâmetros todas as variáveis do serviço (funções incluídas)

bastará assim ao utilizador instanciar o serviço para o seu propósito com base numa escolha critérios de parte (ou de todos) dos parâmetros assim

$$g \triangleq f \ a_1 \ldots a_p$$

é a função especializada que realiza f tendo em conta que $x_1 \ldots x_p$ têm os valores afixados para $a_1 \ldots a_p$

exemplos de uso da avaliação parcial

por exemplo, consideremos a existência do *serviço* "ordenação de listas de valores conforme critério de ordenação"

```
val quicksort (compare: 'a -> 'a -> bool) -> (lista: 'a list)
```

então podemos definir

```
let ord_crescente = quicksort (<)
```

a função `ord_crescente` é uma função de ordenação **crescente** de listas de valores de tipo qualquer ('a)

```
# ord_crescente [4;8;5;7];;
- : int list = [4;5;7;8]
# ord_crescente ['a';'n';'k';'z'];;
- : char list = ['a';'k';'n';'z']
```

uma aplicação popular das expressões lambda em Java decorre do seu uso nas estruturas de dados que se aparentam às listas como as conhecemos na programação funcional: as `stream`

e ao estilo de programação **map-reduce** (i.e. funcional, via composição funcional) que estas possibilitam

```
List<String> umalista =
Arrays.asList("ola", "tudo", "bem", "caro", "colega");

 umalista
.stream()
.filter(s -> s.startsWith("c"))
.map(String::toUpperCase)
.sorted()
.forEach(System.out::println);
```

devolve:

```
CARO
COLEGA
```

```
let starts_with c s = s.[0]=c
let l =  ["ola";"tudo";"bem";"caro";"colega"]

iter print_endline (sort compare
                           (map uppercase_ascii
                              (filter (starts_with 'c') l)))
```

de forma mais elegante, podemos escrever:

```
["ola";"tudo";"bem";"caro";"colega"]
|> filter (start_with 'c')
|> map uppercase_ascii
|> sort compare
|> iter print_endline
```

ambos devolvem:

```
CARO
COLEGA
```

outro exemplo

em Java

```
Arrays.stream(new int[] {1, 2, 3})
   .map(n -> 2 * n + 1)
   .average()
   .ifPresent(System.out::println);
```

```
 5.0
```

o método average é um exemplo de redução de um stream

o método genérico pode ser encontrado neste exemplo

```
String[] myArray = {"I ", "would ",
 "tell ",  "you ", "a ", "joke ",
 "about ", "sodium ", "but ", "Na"};
String result =
  Arrays.stream(myArray)
        .reduce(" ", (a,b) -> a + b);
```

em OCaml

```
open List

let fi =  float_of_int

let media l =
    fi (fold_left (+) 0 l)
    /. (fi (length l))

[1;2;3] |> map (fun n -> 2 * n + 1)
        |> media |> print_float
```

```
 5.
```

ordem de avaliação em stream em Java

é necessário olhar com atenção para a ordem de avaliação das *streams* em Java a documentação oficial diz:

...from the java.util.stream docs

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

Intermediate operations return a new stream. They are always lazy; executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

ordem de avaliação em stream em Java

(retirado de stackoverflow (link))

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares = numbers.stream().filter(n -> {
    System.out.println("filtering " + n);
    return n % 2 == 0;
}).map(n -> {
    System.out.println("mapping " + n);
    return n * n;
}).limit(2).collect(Collectors.toList());


for(Integer i : twoEvenSquares)
{  System.out.println(i); }
```

```
filtering 1 // 1 doesn't pass the filter
filtering 2 // 2 passes the filter, moves on to map
mapping 2 // 2 passes the map and limit steps and is added to output list
filtering 3 // 3 doesn't pass the filter
filtering 4 // 4 passes the filter, moves on to map
mapping 4 // 4 passes the map and limit steps and is added to output list
```

fechos

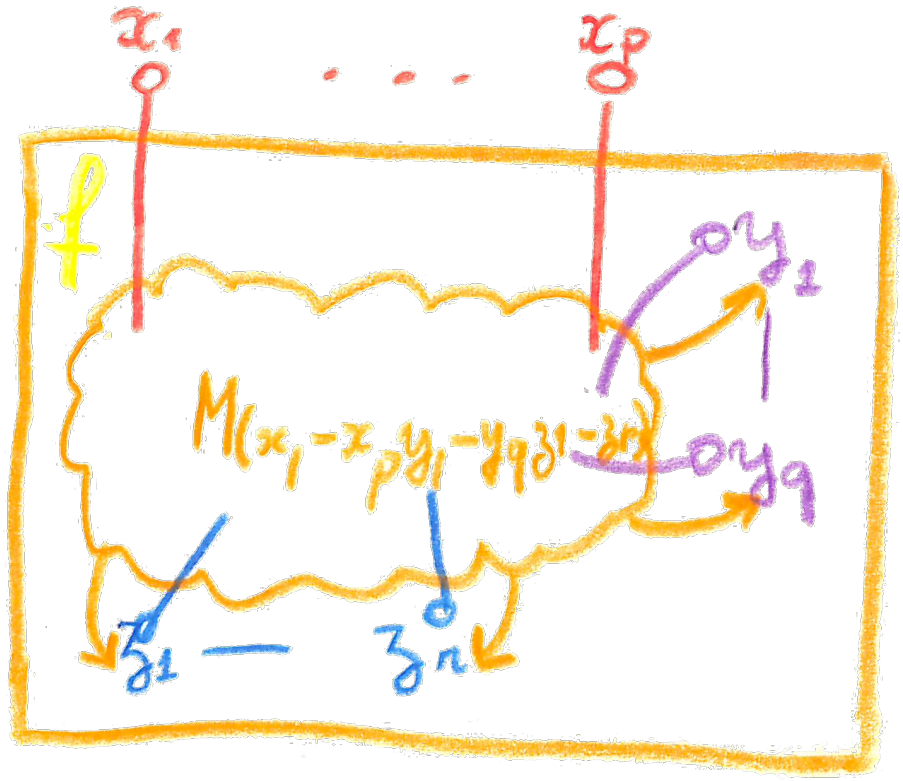
uma viagem pelos mecanismos de execução de um programa

para perceber melhor alguns fenómenos ligados ao suporte das funções dadas ao programador, vamos olhar como elas são representadas e executadas ...

quer seja nativamente quer por uma máquina virtual

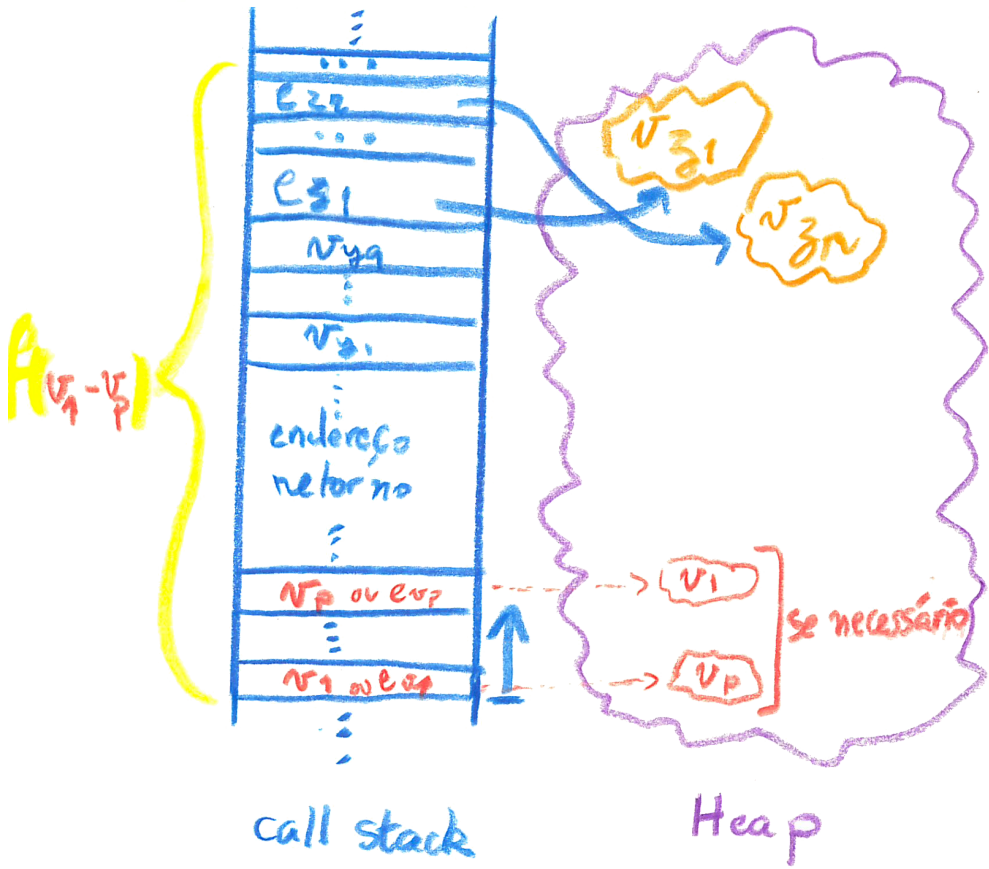
representação memória da chamada de *f*

let f $x_1 \cdots x_p$ =
  let y_1 = valor simples in
  ...
  let y_q = valor simples in
  let z_1 = valor complexo in
  ...
  let z_n = valor complexo in
    $M(x_1 - x_p\ y_1 - y_q\ z_1 - z_n)$



com um pouco mais de realismo: na pilha de chamada e com dados na heap

let f $x_1 \cdots x_p$ =
  let y_1 = valor simples in
  ...
  let y_q = valor simples in
  let z_1 = valor complexo in
  ...
  let z_n = valor complexo in
    $M(x_1 - x_p\ y_1 - y_q\ z_1 - z_n)$



representação memória da chamada de função

na verdade, a situação concreta é ainda um pouco mais complexa

analisemos o seguinte exemplo (em OCaml, mas pode ser descrito em qualquer outra linguagem de programação que tenha funções de primeira classe)

```
let f x =
  let y = read_int () in
  let g z = x + y + 2 * z in
   g
```

o que devolve $(f\ 7)$ neste exemplo?
que fenómeno aqui surge? que cautela aparece?

outro exemplo ilustrativo é (em Java, desta vez):

```
int m = 4;
IntUnaryOperator f = a -> a + m;
m = 6;
out.println(f.apply(2));
```

qual é o resultado em output?

chamadas terminais

programação por continuação