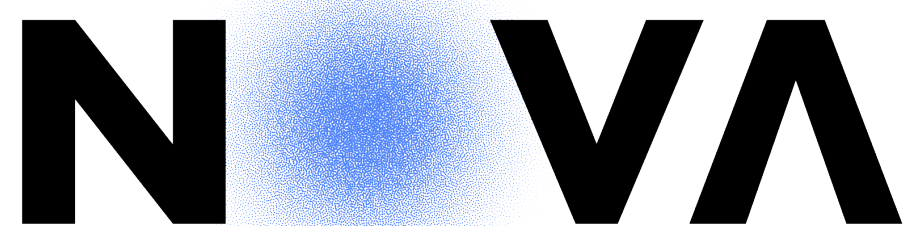


Linguagens e Ambientes de Programação (Aula Teórica 7)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Instruções de submissão

- Classroom assignment: <https://classroom.github.com/a/U5y0mL-Z>
- Aceitar o assignment e criar um grupo (um elemento)
- Juntar-se a uma equipa já criada (o outro elemento)
- Nome do Grupo:
 - PX-12345-PY-54321
 - PX, PY - turnos
 - 12345, 54321 - números de aluno
 - POR ORDEM CRESCENTE DE NÚMERO!!



Agenda

- Tipos compostos: pares, tipos soma e pattern matching.

Abstração

- As linguagens de programação têm mecanismos internos de construção e extensão das suas operações e tipos de dados.
 - Abstração funcional (funções): estendem as operações básicas da linguagem
 - Abstracção de tipos (polimorfismo): estendem a aplicação de operações a valores de diferentes naturezas
 - Abstração de dados (tipos compostos): estendem os valores (e tipos) disponíveis para exprimir computações
- Tipos primitivos: inteiros, reais, booleanos, strings, caracteres, ...
- Tipos compostos: pares (tuplos), registos, variantes, listas, tipos algébricos

O tipo “tuplo”

- O tipo par corresponde à **conjunção** na correspondência Curry-Howard.
- Corresponde à definição do produto cartesiano de dois (ou mais) conjuntos na correspondência entre tipos e conjuntos.
- Declara o tipo dos valores **que têm duas** (ou mais) **partes** de tipos (potencialmente) diferentes.

$$A \times B$$

- O tipo tuplo primitivo admite um número finito de componentes, cada componente pode ser de qualquer tipo.
- O tipo tuplo permite combinar valores de forma heterogénea, em oposição às estruturas de dados tradicionais que são coleções homogéneas.

Introdução do tipo tuplo

```
▷ (1,2)  
[4] ✓ 0.0s  
... - : int * int = (1, 2)
```

```
▷ ("The answer is", 42)  
[5] ✓ 0.0s  
... - : string * int = ("The answer is", 42)
```

```
▷ ("FullName", ("John", "Doe"))  
[9] ✓ 0.0s  
... - : string * (string * string) = ("FullName", ("John", "Doe"))
```

$$\frac{A \quad B}{A \wedge B}$$

Introdução do tipo tuplo

```
▷ let pairup = fun x y -> (x,y);;
```

```
pairup "Hello" (pairup 1 2)
```

[11] ✓ 0.0s

```
... val pairup : 'a -> 'b -> 'a * 'b = <fun>
```

```
... - : string * (int * int) = ("Hello", (1, 2))
```

$$\frac{A \quad B}{A \wedge B}$$

```
▷ let rotate_point (x, y) theta =
```

```
  let cos_theta = cos theta in
```

```
  let sin_theta = sin theta in
```

```
  let new_x = x *. cos_theta -. y *. sin_theta in
```

```
  let new_y = x *. sin_theta +. y *. cos_theta in
```

```
  (new_x, new_y)
```

[7] ✓ 0.0s

```
... val rotate_point : float * float -> float -> float * float = <fun>
```

```
let rotated_point = rotate_point (3.0, 4.0) (Float.pi /. 4.0)
```

[8] ✓ 0.0s

```
... val rotated_point : float * float = (-0.707106781186547, 4.94974746830583268)
```

Eliminação do tipo tuplo

```
▷ let p = (1,2) in fst p + snd p
```

```
[15] ✓ 0.0s
```

```
... - : int = 3
```

```
▷ let (x,y) = (1,2) in x + y
```

```
[16] ✓ 0.0s
```

```
... - : int = 3
```

```
▷ (fun (x,y) -> x + y ) (1,2)
```

```
[17] ✓ 0.0s
```

```
... - : int = 3
```

$$\frac{A \wedge B}{A}$$

$$\frac{A \wedge B}{B}$$

Definição de tipos por nome

- Os tipos compostos tornam-se “reutilizáveis” se associados a um nome novo.
- Com inferência de tipos, este aspeto só é particularmente importante em aplicações grandes e/ou se quisermos ter tipos “opacos” em módulos.
- Mais tarde voltaremos à definição de módulos e assinaturas de módulos.
- Também são importantes para declarar registos.

```
▷ type coordinates = float * float
[33] ✓ 0.0s
... type coordinates = float * float
```

```
▷ module type PointsSig = sig
  type coordinates
  val create_point : float -> float -> coordinates
  val rotate_point : coordinates -> float -> coordinates
end

module PointsImpl = struct
  type coordinates = float * float

  let rotate_point (point: coordinates) theta =
    let cos_theta = cos theta in
    let sin_theta = sin theta in
    let new_x = (fst point) *. cos_theta -. (snd point) *. sin_theta in
    let new_y = (fst point) *. sin_theta +. (snd point) *. cos_theta in
    (new_x, new_y)
end
[34] ✓ 0.0s
```

Registos

- Registos são tipos compostos onde os componentes são acedidos por nome.

```
type person = {name: string; age: int}
```

[33] ✓ 0.0s

```
... type person = { name : string; age : int; }
```

```
let p = {name = "John"; age = 42}
```

[35] ✓ 0.0s

```
... val p : person = {name = "John"; age = 42}
```

```
"The person's name is " ^ p.name ^ " and age is " ^ string_of_int p.age
```

[37] ✓ 0.0s

```
... - : string = "The person's name is John and age is 42"
```

```
fun {name;age} -> "The person's name is " ^ name ^ " and age is " ^ string_of_int age
```

[13] ✓ 0.0s

```
... - : person -> string = <fun>
```

Funções com vários parâmetros e vários resultados

- A forma tradicional de chamar uma função, em que todos os parâmetros são dados de uma só vez, faz-se com tuplos em oCaml.
- Os tuplos também permitem o retorno de vários valores simultaneamente.
- À capacidade de chamar uma função parcialmente chama-se Currying.

```
▷ let rotate_point (p, theta) =  
  let (x,y) = p in  
  let cos_theta = cos theta in  
  let sin_theta = sin theta in  
  let new_x = x *. cos_theta -. y *. sin_theta in  
  let new_y = x *. sin_theta +. y *. cos_theta in  
  (new_x, new_y)  
[19] ✓ 0.0s  
... val rotate_point : (float * float) * float -> float * float = <fun>  
  
let point = (3.0, 4.0) in  
let rotated_point = rotate_point (point, (Float.pi /. 4.0))  
[17] ✓ 0.0s  
... val rotated_point : float * float = (-0.707106781186547, 4.94974746830583268)
```

O tipo “soma” (variantes ou enumerados)

- O tipo par corresponde à **disjunção** na correspondência Curry-Howard.
- Corresponde à definição da união (etiquetada) de dois (ou mais) conjuntos na correspondência entre tipos e conjuntos.
- Declara o tipo dos valores que **têm uma de duas** (ou mais) **partes** de tipos (potencialmente) diferentes.

$$A + B$$

- O tipo soma primitivo admite um número finito de alternativas, onde cada alternativa pode ser de qualquer tipo.
- O tipo soma corresponde a combinar valores diferentes de forma heterogénea.

Introdução do tipo soma

- Um tipo soma assume várias alternativas possíveis para os seus valores. É uma forma de polimorfismo ad-hoc.

```
▷ type species = Dog | Cat | Bird | Fish  
  type pet = { name: string; species: species; age: int }
```

[20] ✓ 0.0s

```
... type species = Dog | Cat | Bird | Fish
```

```
... type pet = { name : string; species : species; age : int; }
```

```
▷ let p = {name = "Fido"; species = Dog; age = 3}
```

[]

$$\frac{A}{A \vee B}$$
$$\frac{B}{A \vee B}$$

Introdução do tipo soma (com dados)

```
▷ type point = float * float  
  
type figure =  
  | Circle of point * float  
  | Rectangle of point * point  
  | Triangle of point * point * point
```

[41] ✓ 0.0s

... type point = float * float

... type figure =
 Circle of point * float
 | Rectangle of point * point
 | Triangle of point * point * point

```
▷ Circle ((3.0, 4.0), 2.0)
```

[42] ✓ 0.0s

... - : figure = Circle ((3., 4.), 2.)

$$\frac{A}{A \vee B}$$
$$\frac{B}{A \vee B}$$

Introdução do tipo soma (com dados)

```
[23] type 'a option = None | Some of 'a
✓ 0.0s
... type 'a option = None | Some of 'a

▷
[25] Some "Dwarf Knight"
✓ 0.0s
... - : string option = Some "Dwarf Knight"
```

$$\frac{A}{A \vee B}$$
$$\frac{B}{A \vee B}$$

Eliminação do tipo soma: Pattern matching!

- Um valor de um tipo soma pode ser de uma de um conjunto de alternativas.

$$\frac{A \implies C \quad B \implies C \quad A \vee B}{C}$$

- Só é possível progredir de forma segura por análise de casos detalhada.
- Todos os ramos têm que ter o mesmo tipo, como na expressão condicional.

```
let describe f =  
  match f with  
  | Circle (x,y,r) -> "Circle at " ^ string_of_float x ^ "," ^ string_of_float y  
  | Rectangle (x1,y1) -> "Rectangle at " ^ string_of_float (fst x1) ^ "," ^ string_of_float (snd x1)  
  | Triangle (x1,y1,z1) -> "Triangle at " ^ string_of_float (fst x1) ^ "," ^ string_of_float (snd x1)
```

[39] ✓ 0.0s

```
... val describe : figure -> string = <fun>
```


Variantes em C (Unions)

```
enum ShapeType {
    CIRCLE,
    RECTANGLE,
    TRIANGLE
};

union GeometricShape {
    enum ShapeType type;

    struct {
        double x;
        double y;
        double radius;
    } circle;

    struct {
        double x1;
        double y1;
        double x2;
        double y2;
    } rectangle;

    struct {
        double x1;
        double y1;
        double x2;
        double y2;
        double x3;
        double y3;
    } triangle;
};
```

```
union GeometricShape circle;
circle.type = CIRCLE;
circle.circle.x = 2.0;
circle.circle.y = 3.0;
circle.circle.radius = 5.0;
```

```
union GeometricShape rectangle;
rectangle.type = RECTANGLE;
rectangle.rectangle.x1 = 1.0;
rectangle.rectangle.y1 = 2.0;
rectangle.rectangle.x2 = 6.0;
rectangle.rectangle.y2 = 5.0;
```

```
union GeometricShape triangle;
triangle.type = TRIANGLE;
triangle.triangle.x1 = 1.0;
triangle.triangle.y1 = 1.0;
triangle.triangle.x2 = 4.0;
triangle.triangle.y2 = 5.0;
triangle.triangle.x3 = 7.0;
triangle.triangle.y3 = 2.0;
```

Case Classes e Pattern Matching em Scala

```
sealed trait class GenericShape

case class Circle(x: Double, y: Double, radius: Double) extends GenericShape
case class Rectangle(x1: Double, y1: Double, x2: Double, y2: Double) extends GenericShape
case class Triangle(x1: Double, y1: Double, x2: Double, y2: Double, x3: Double, y3: Double)
extends GenericShape

def printShapeDetails(shape: GeometricShape): Unit =
  shape match
    case Circle(x, y, r) => println(s"Circle: Center ($x, $y), Radius $r")
    case Rectangle(x1, y1, x2, y2) =>
      println(s"Rectangle: Top-left ($x1, $y1), Bottom-right ($x2, $y2)")
    case Triangle(x1, y1, x2, y2, x3, y3) =>
      println(s"Triangle: Vertex 1 ($x1, $y1), Vertex 2 ($x2, $y2), Vertex 3 ($x3, $y3)")
```

Pattern matching

```
let whatDoYouHave o =  
  match o with  
  | None -> "Nothing"  
  | Some x -> "I have "^x
```

[]

```
let species_of p =  
  match p.species with  
  | Dog -> "Dog"  
  | Cat -> "Cat"  
  | Bird -> "Bird"  
  | Fish -> "Fish"
```

[]

Pattern matching

```
▷ type point = float * float

type figure =
  | Circle of point * float
  | Rectangle of point * point
  | Triangle of point * point * point
  | Polygon of point list
```

[41] ✓ 0.0s

```
▷ let how_may_points_in f =
  match f with
  | Circle _ -> 0
  | Rectangle _ -> 4
  | Triangle _ -> 3
```

[46] ✓ 0.0s

... File "[46]", lines 2-5, characters 2-19:

```
2 | ..match f with
3 |   | Circle _ -> 0
4 |   | Rectangle _ -> 4
5 |   | Triangle _ -> 3
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a case that is not matched:

Polygon _

... val how_may_points_in : figure -> int = <fun>

Pattern matching

```
▷ type point = float * float

type figure =
| Circle of point * float
| Rectangle of point * point
| Triangle of point * point * point
| Polygon of point list
```

[41] ✓ 0.0s

```
▷ let how_many_points_in f =
  match f with
  | Circle _ -> 0
  | Rectangle _ -> 4
  | Triangle _ -> 3
  | Polygon points -> List.length points
```

```
▷ let do_you_like_this_figure f =
  match f with
  | Circle _ -> "Yes"
  | _ -> "No"
```

[]