

# Linguagens e Ambientes de Programação (Aula Teórica 12)

**LEI - Licenciatura em Engenharia Informática**

**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**



NOVA SCHOOL OF  
SCIENCE & TECHNOLOGY

# Agenda

---

- Análise de resultados projetos
- Análise da resolução do teste
- Sistema de Módulos

# Sistema de Módulos

---

- Espaços de nomes
  - Grupos de declarações (normalmente) relacionadas isolados de outros grupos por via da qualificação dos nomes num módulo.
  - Permite a reutilização dos mesmos nomes em contextos diferentes sem colisões.
  - Pacotes e classes em Java, ficheiros/módulos em C, estruturas/módulos em ocaml
- Abstração
  - Permite esconder/revelar selectivamente informação (*information hiding*)
  - Isolamento de código, melhor desenvolvimento e manutenção, ownership, etc.
- Reutilização de código
  - reutilização sem cópia, modularidade, (cf. herança em Java)
- (em OCaml) Parametrização de módulos
  - Os Functores em OCaml são como funções de módulos para módulos (cf. traits em Scala)

# Módulos em OCaml

- Os módulos são definidos por estruturas (**struct**)
- Os tipos para os módulos são assinaturas (**sig**)
- As definições de tipos por omissão são públicas (**type**)
- As implementações dos nomes ficam privadas (**val**)



```
module MyList = struct
  type 'a list = Nil | Cons of 'a * 'a list
  let empty = Nil
  let rec length = function
    | Nil → 0
    | Cons (_, xs) → 1 + length xs
  let insert x xs = Cons (x, xs)
  let head = function
    | Nil → None
    | Cons (x, _) → Some x
  let tail = function
    | Nil → None
    | Cons (_, xs) → Some xs
end
```

[7] ✓ 0.0s

```
... module MyList :
  sig
    type 'a list = Nil | Cons of 'a * 'a list
    val empty : 'a list
    val length : 'a list → int
    val insert : 'a → 'a list → 'a list
    val head : 'a list → 'a option
    val tail : 'a list → 'a list option
  end
```

# Espaço de nomes

- Os nomes declarados num módulo podem ser usados de forma qualificada (com o nome do Módulo e um ponto: **List.fold\_right**)
- Ou pode-se usar a directiva **open** para expandir os nomes do módulo usado no módulo cliente.
- o módulo **StdLib** está sempre aberto.

```
module MyStack = struct
  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let push x xs = MyList.insert x xs
  let pop = MyList.tail
  let top = MyList.head
end

[17] ✓ 0.0s

... module MyStack :
  sig
    type 'a stack = 'a MyList.list
    val empty : 'a MyList.list
    val push : 'a → 'a MyList.list → 'a MyList.list
    val pop : 'a MyList.list → 'a MyList.list option
    val top : 'a MyList.list → 'a option
  end
```

```
module MyStack = struct
  open MyList

  type 'a stack = 'a list
  let empty = empty
  let push x xs = insert x xs
  let pop = tail
  let top = head
end

[14] ✓ 0.0s

... module MyStack :
  sig
    type 'a stack = 'a MyList.list
    val empty : 'a MyList.list
    val push : 'a → 'a MyList.list → 'a MyList.list
    val pop : 'a MyList.list → 'a MyList.list option
    val top : 'a MyList.list → 'a option
  end
```



# Abstração de nomes

- Os tipos dos módulos permitem ainda esconder a definição dos tipos
- Uma assinatura pode ter várias implementações compatíveis (opacas)

```
module type Stack =  
  sig  
    type 'a stack  
    val empty : 'a stack  
    val push : 'a → 'a stack → 'a stack  
    val pop : 'a stack → 'a stack option  
    val top : 'a stack → 'a option  
  end  
  
module MyStack : Stack  
  
module AnotherStack : Stack
```

```
module type Stack = sig  
  type 'a stack  
  val empty : 'a stack  
  val push : 'a → 'a stack → 'a stack  
  val pop : 'a stack → 'a stack option  
  val top : 'a stack → 'a option  
end
```

```
module MyStack:Stack = struct  
  type 'a stack = 'a MyList.list  
  let empty = MyList.empty  
  let push x xs = MyList.insert x xs  
  let pop = MyList.tail  
  let top = MyList.head  
end
```

```
module AnotherStack : Stack = struct  
  type 'a stack = 'a list  
  let empty = []  
  let push x xs = x :: xs  
  let pop = function  
    | [] → None  
    | _ :: xs → Some xs  
  let top = function  
    | [] → None  
    | x :: _ → Some x  
end
```

# Tipos e nomes

- A especialização de tipos de módulos pode ser feita com um módulo de adaptação.



```
module IntStack = (struct
  type stack = int MyStack.stack
  let empty = MyStack.empty
  let push = MyStack.push
  let pop = MyStack.pop
  let top = MyStack.top
end : sig
  type stack
  val empty : stack
  val push : int → stack → stack
  val pop : stack → stack option
  val top : stack → int option
end)
```

[36] ✓ 0.0s

```
... module IntStack :
  sig
    type stack
    val empty : stack
    val push : int → stack → stack
    val pop : stack → stack option
    val top : stack → int option
  end
```

# Módulos e ficheiros

- A organização em ficheiros separa a estrutura (struct) da assinatura (sig)
- Ficheiros MyList.ml, MyStack.mli, MyStack.ml

```
s > LAP 2024-12 > MyList.ml > ...
type 'a list = Nil | Cons of 'a * 'a list
'a list
let empty = Nil
'a list -> int
let rec length = function
| Nil -> 0
| Cons (_, xs) -> 1 + length xs
'a -> 'a list -> 'a list
let insert x xs = Cons (x, xs)
'a list -> 'a option
let head = function
| Nil -> None
| Cons (x, _) -> Some x
'a list -> 'a list option
let tail = function
| Nil -> None
| Cons (_, xs) -> Some xs
```

```
> LAP 2024-12 > MyStack.mli > ...
type 'a stack
val empty : 'a stack
val push : 'a -> 'a stack -> 'a stack
val pop : 'a stack -> 'a stack option
val top : 'a stack -> 'a option
```

```
s > LAP 2024-12 > MyStack.ml > ...
type 'a stack = 'a MyList.list
'a
let empty = MyList.empty
'a -> 'b -> 'c
let push x xs = MyList.insert
'a
let pop = MyList.tail
'a
let top = MyList.head
```



# Módulos e Funtores (funções de módulos para módulos)



```
module type X = sig
|   val x : int
end

module IncX (M : X) = struct
|   let x = M.x + 1
end
```

[23]

✓ 0.0s

... module type X = sig val x : int end

... module IncX : functor (M : X) → sig val x : int end

# Módulos e Funtores (funções de módulos para módulos)

```
▷ module type X = sig
  type t
end

module Stack = struct
  module Make (M : X) = struct
    type stack = M.t list
    let empty = []
    let push x xs = x :: xs
    let pop = function
      | [] → None
      | _ :: xs → Some xs
    let top = function
      | [] → None
      | x :: _ → Some x
    end
  end
end

module IntStack = Stack.Make (struct type t = int end)

let s = IntStack.empty
let _ = assert (IntStack.top s = None)
let _ = assert (IntStack.top (IntStack.push 1 s) = Some (1))
let _ = assert (IntStack.pop (IntStack.push 1 s) = Some (IntStack.empty))
```

[71] ✓ 0.0s

# Módulos e Funtores (funções de módulos para módulos)



```
module Pair = struct
  type t = int * string
  let compare (x1, y1) (x2, y2) =
    if x1 < x2 then -1
    else if (x1 = x2 && y1 < y2) then -1
    else if (x1 = x2 && y1 = y2) then 0 else 1
end
```

```
module Str = Set.Make(Pair)
```

```
let s = Str.empty
let s = Str.add (1, "a") s
let s = Str.add (2, "b") s
let s = Str.add (3, "c") s
let s = Str.add (4, "d") s
let _ = assert Str.(mem (1, "a") s)
```

[83]

✓ 0.0s