Linguagens e Ambientes de Programação (Aula Teórica 15)

LEI - Licenciatura em Engenharia Informática

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

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda

- Sistema de Módulos
- Functores
- Concorrência
  - Promessas

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 MyModule = struct
  type primary_color = Red | Green | Blue

  let inc x = x + 1
  let dec x = x - 1
end
```

```
(* Inferred signature *)

module MyModule :
  sig
    type primary_color = Red | Green | Blue
    val inc : int -> int
    val dec : int -> int
 end
```

utop

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 M = struct
  let x = 42
end

M.x                     (* 42 *)
x                       (* Unbound value x *)

let y = M.(x * x + x) (* val y : int = 1806 *)

open M
x                       (* 42 *)
```

Módulo `MyList`

```
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
```

```
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
```

Java vs. Ocaml

- Java

```
s = new List();
s.insert(1);
```

- OCaml

```
let s  = MyList.empty;;
let s' = MyList.insert 6 s;;
```

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 is_empty : 'a stack -> bool
  val push     : 'a -> 'a stack -> 'a stack
  val top      : 'a stack -> 'a option
  val pop      : 'a stack -> 'a stack option
end
```

```
module ListStack : Stack = struct
  type 'a stack = 'a list
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let top = function
    | [] -> None
    | x :: _ -> Some x
  let pop = function
    | [] -> None
    | _ :: xs -> Some xs
end
```

```
module MyListStack : Stack = struct
  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let is_empty = MyList.is_empty
  let push x s = MyList.insert x s
  let top = MyList.head
  let pop = MyList.tail
end;;
```

Espaço de nomes (novamente)

```
module MyListStack : Stack = struct


  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let is_empty = MyList.is_empty
  let push x s = MyList.insert x s
  let top = MyList.head
  let pop = MyList.tail
end;;
```

```
module MyListStack : Stack = struct
  open MyList

  type 'a stack = 'a list
  let empty = empty
  let is_empty = is_empty
  let push x s = insert x s
  let top = head
  let pop = tail
end;;
```

Módulo ListStackCachedSize

- Implementar e testar!

```
module ListStackCachedSize : Stack = struct
  type 'a stack = 'a list * int

  let empty = ([], 0)
  let is_empty s =
    match s with
    | ([], _) -> true
    | _ -> false
  let push x s = (x::(fst s), (snd s)+1)
  let top s = match s with
    | ([], _) -> None
    | (x :: xs, _) -> Some x
  let pop s = match s with
    | ([], _) -> None
    | (x :: xs, n) -> Some (xs, n-1)
  let size s = snd s
end
```

Módulo Counter com Refs!

- Implementar e testar!

```
module type Counter = sig
  type t
  (** [create v] makes a new counter the initial value [v]. *)
  val create : int -> t

  (** [inc c] increments the counter by 1. *)
  val inc    : t -> unit

  (** [dec c] decrements the counter by 1. *)
  val dec    : t -> unit

  (** [get c] returns the current value. *)
  val get    : t -> int

  (** [reset c] sets the counter to zero. *)
  val reset  : t -> unit
end
```

```
module CounterRef : Counter = struct
  type t = int ref

  let create v = ref v

  let inc c = c := !c + 1

  let dec c = c := !c - 1

  let get c = !c

  let reset c = c := 0

end
```

Tipos e nomes

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

```
module IntStack = (struct
  (* 1. Build on a generic "ListStack" module
        by fixing its element type to int. *)
  type stack = int MyListStack.stack

  (* 2. Re-export the operations from ListStack *)
  let empty = MyListStack.empty
  let push  = MyListStack.push
  let pop   = MyListStack.pop
  let top   = MyListStack.top
end : sig
  (* 3. Users of IntStack only see the abstract type [stack]
        and the four operations with the following types *)
  type stack
  val empty : stack
  val push  : int -> stack -> stack
  val pop   : stack -> stack option
  val top   : stack -> int option
end)
```

```
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, Stack.mli, MyStackList.ml

LAP-2025 > MyList.ml > ...

```
type 'a list = Nil | Cons of 'a * 'a list

let empty = Nil

let is_empty = function -> true | _ -> false

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
```

LAP-2025 > Stack.mli > ...

```
type 'a stack
val empty    : 'a stack
val is_empty : 'a stack -> bool
val push     : 'a -> 'a stack -> 'a stack
val top      : 'a stack -> 'a option
val pop      : 'a stack -> 'a stack option
end
```

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 Functores (funções de módulos para módulos)

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

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

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

module IncX : functor (M : X) -> sig val x : int end
```

Módulos e Functores (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

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)
```

```
module Stack :
  sig
    module Make :
      functor (M : X) ->
        sig
          type stack = M.t list
          val empty : 'a list
          val push : 'a -> 'a list -> 'a list
          val pop : 'a list -> 'a list option
          val top : 'a list -> 'a option
        end
  end
```

Módulos e Functores (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)
```

Concurrência

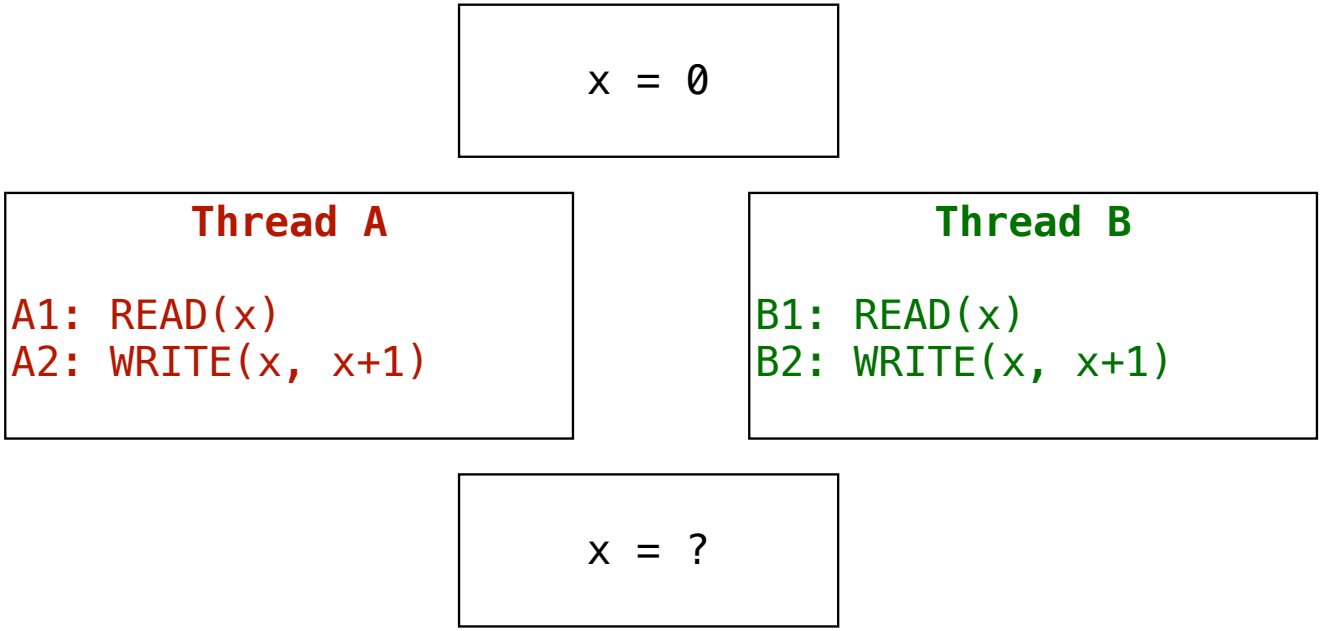
Concorrência

- Capacidade de realizar várias computações que se sobrepõem no tempo, e não exigir que sejam feitas em sequência.
  - Interfaces gráficos; para assegurar respostas rápida a acções do utilizador.
  - Folhas de cálculo; para recalcular todos os cálculos sem interrupções.
  - Web Browser; para carregar e mostrar páginas de forma incremental.
  - Servidores; para atender vários clientes sem os fazer esperar.
- Como?
  - Entrelaçamento: comutando entre as várias computações ativas rapidamente.
  - Paralelismo: utilizando vários processadores físicos (multi-core).
- Preemptive / Collaborative

Não determinismo

- Um programa que pode ter resultados diferentes de cada vez que é executado é um programa não determinista.
- A introdução de concorrência causa não determinismo, ao não fixar a ordem pela qual operações são executadas.
- Quando dois programas imperativos partilham variáveis de estado, as possibilidades de alteração desse mesmo estado são indeterminadas. Logo, não é fácil prever o comportamento exacto dos programas.
- As interações/interferências entre programas são benignas ou malignas (race conditions)
- As linguagens funcionais tornam mais fácil o raciocínio sobre programas porque uma expressão pura denota sempre o mesmo valor.

Não determinismo



Promessas ou Futuros

- Representam computações que ainda não acabaram mas que irão denotar um valor algures num instante futuro (diferido).
- Normalmente associadas a uma computação concorrente ao thread principal.

```
async function getNames() {
  const response = await fetch('https://server.com/users')
  const data = await response.json()
  return data.map(user ⇒ user.name)
}
```

- Async (Jane Street) e Lwt (Ocsigen) são duas bibliotecas populares para implementar computação assíncrona em OCaml.

Promessas ao estilo Lwt

- São abstrações de dados para um modelo de computação assíncrona.

- As promessas são referências, o seu valor pode mudar.
- Quando é criada não contém nada.
- Uma promessa pode ser cumprida e preenchida por um valor
- Uma promessa pode ser rejeitada (preenchida por uma exceção)
- Em ambos os casos diz-se resolvida.

Assinatura do Módulo Promessa

```
module type PROMISE = sig
  type 'a state = Pending | Fulfilled of 'a | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  val make : unit -> 'a promise * 'a resolver

  (** [return x] is a new promise that is already fulfilled with value [x]. *)
  val return : 'a -> 'a promise

  (** [state p] is the state of the promise *)
  val state : 'a promise -> 'a state

  (** [fulfill r x] fulfills the promise [p] associated with [r] with value [x], meaning that
      [state p] will become [Fulfilled x].  Requires: [p] is pending. *)
  val fulfill : 'a resolver -> 'a -> unit

  (** [reject r x] rejects the promise [p] associated with [r] with exception [x], meaning that
      [state p] will become [Rejected x].  Requires: [p] is pending. *)
  val reject : 'a resolver -> exn -> unit
end
```

Implementação do Módulo Promessa

```
module Promise : PROMISE = struct
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise = 'a state ref

  type 'a resolver = 'a promise

  (** [write_once p s] changes the state of [p] to be [s]. If [p] and [s] are both pending,
      that has no effect. Raises: [Invalid_arg] if the state of [p] is not pending. *)
  let write_once p s =
    if !p = Pending then p := s else invalid_arg "cannot write twice"

  let make () = let p = ref Pending in (p, p)

  let return x = ref (Fulfilled x)

  let state p = !p

  let fulfill r x = write_once r (Fulfilled x)

  let reject r x = write_once r (Rejected x)
end
```

Uso de Promessas

```
(** Lets see how to use promises *)
let compute_fact_sync n : int Promise.promise =
  let p, r = Promise.make () in
  begin
    try
      let result = fact n in      (* pure factorial function *)
      Promise.fulfill r result    (* settle with the computed value *)
    with exn ->
      Promise.reject r exn        (* an error, which never happens here*)
  end;
  p

let check name p =
  match Promise.state p with
  | Pending ->
      Printf.printf "%s is still computing…\n" name
  | Fulfilled v ->
      Printf.printf "%s = %d\n" name v
  | Rejected exn ->
      Printf.printf "%s failed\n" name

let p5 = compute_fact_sync 5 in check "5!" p5
```

Uso de Promessas em LWT

```
(* Pure-function *)
let rec fact n =
  if n <= 1 then 1 else n * fact (n - 1)

(* Initialize the preemptive pool with default bounds (min=0, max=4) *)
let () = Lwt_preemptive.simple_init ()

(* Offload [fact n] onto Lwt's preemptive pool *)
let compute_fact n : int Lwt.t =
  Lwt_preemptive.detach fact n

let () =
  let work = [
    compute_fact 20 >>= fun r -> Lwt_io.printf "20! = %d\n" r;
    compute_fact 25 >>= fun r -> Lwt_io.printf "25! = %d\n" r;
  ] in
  Lwt_main.run (Lwt.join work)
```

No utop:

```
#require "lwt.unix";;
#open Lwt;;
#open Lwt.Infix;;
```