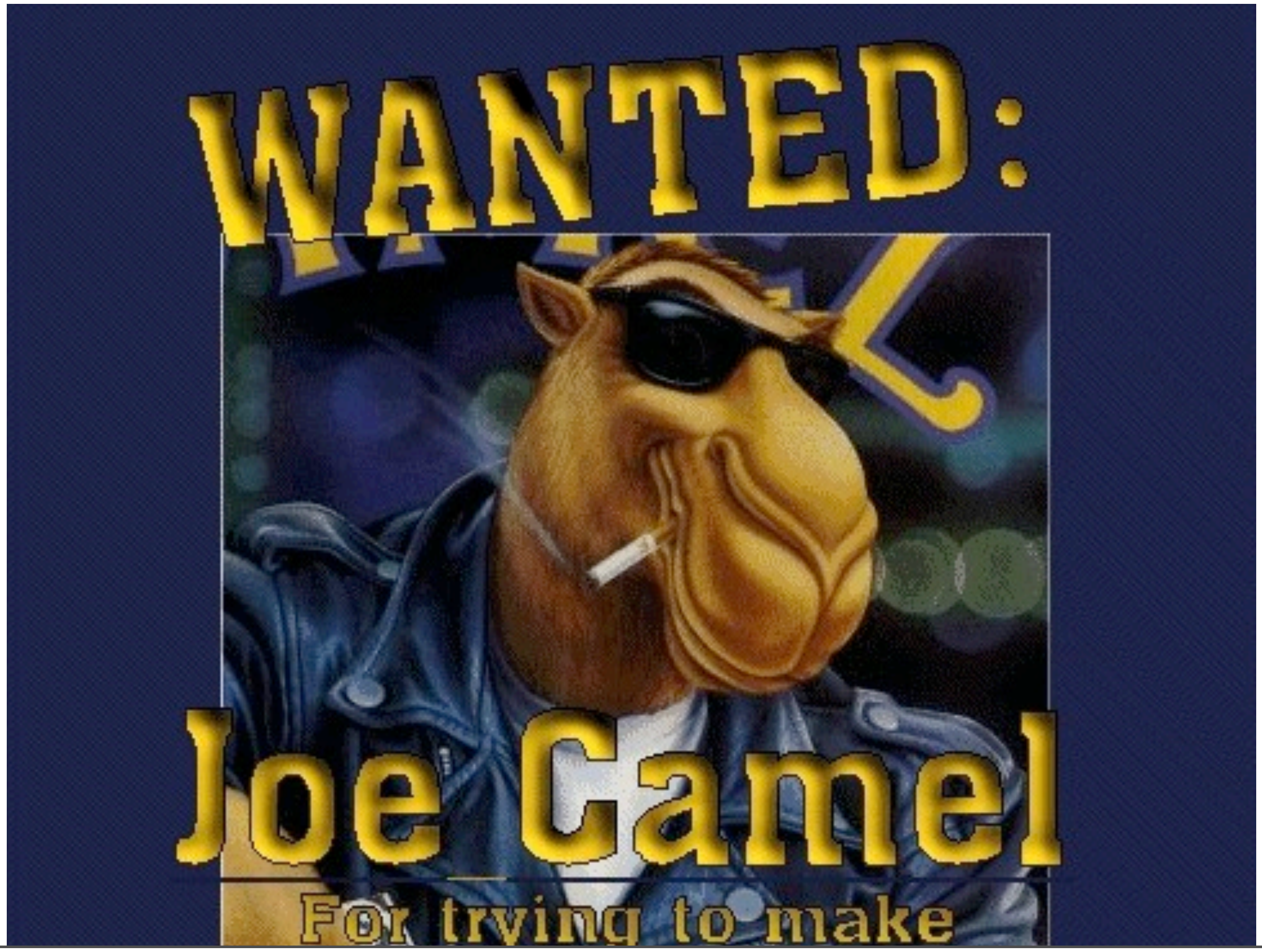
J-O-Caml (5)

jean-jacques.levy@inria.fr
pauillac.inria.fr/~levy/qinghua/j-o-caml

Qinghua, December 3



Plan of this class

- solution of exercices
- modules and signatures
- parameterised modules, functors
- manipulating terms as AST (moved to next lecture)
- duality between objects and modules
- primitives for concurrency
- ariane 5 story (end)
- overall structure of labeling program

Modules and signatures

- several syntactic forms

```
module M : sig ... end = struct ... end

module M = (struct ... end : sig ... end)

module type T = sig ... end

module M = (M’ : T)
```

Modules and Signatures

```
module Fifo : sig
    type 'a t
    exception Empty_Fifo
    val create : unit -> 'a t
    val add : 'a t -> 'a -> unit
    val take : 'a t -> 'a
    val iter : ('a -> unit) -> 'a t -> unit
end = struct
  type 'a t = {mutable hd : 'a list; mutable tl: 'a list }
  (* hd points to fst of queue; tl points to last of queue *)
  exception Empty_Fifo
  let create () = {hd = [ ]; tl = [ ] }
  let add  f x = match f.hd, f.tl with
    | [ ], [ ] -> f.tl <- [x]; f.hd <- f.tl
    | _ , _ -> f.tl <- [x]; f.hd <- f.hd @ f.tl
  let take f = match f.hd with
    | [ ] -> raise Empty_Fifo
    | [ x ] -> f.hd <- [ ]; f.tl <- [ ]; x
    | x :: f' -> f.hd <- f'; x
  let iter f fifo = List.iter f fifo.hd
end ;;
```

Exercices

- FIFO as circular buffer [careful with duality allocation -- polymorphism!]

```
module Fifo1 : sig
    type 'a t
    exception Empty_Fifo
    exception Full_Fifo
    val make : int -> 'a -> 'a t
    val add : 'a t -> 'a -> unit
    val take : 'a t -> 'a
    val iter : ('a -> unit) -> 'a t -> unit
end = struct
  type 'a t = {mutable hd: int; mutable tl: int; mutable full: bool; mutable empty: bool;
               content: 'a array}
  exception Empty_Fifo
  exception Full_Fifo
  let make n x = {hd = 0; tl = 0; full = false; empty = true; content = Array.make n x}
  let add f x  = if f.full then raise Full_Fifo else begin
    f.content.(f.tl) <- x; f.tl <- f.tl + 1; if f.tl >= Array.length f.content then f.tl <- 0
;
    f.empty <- false; f.full <- f.tl = f.hd
  end
  let take f = if f.empty then raise Empty_Fifo else
    let res = f.content.(f.hd) in
    f.hd <- f.hd + 1; if f.hd >= Array.length f.content then f.hd <- 0 ;
    f.empty <- f.tl = f.hd; f.full <- false;
    res
  let iter f fifo = Array.iter f fifo.content
end ;;
```

Parameterized modules

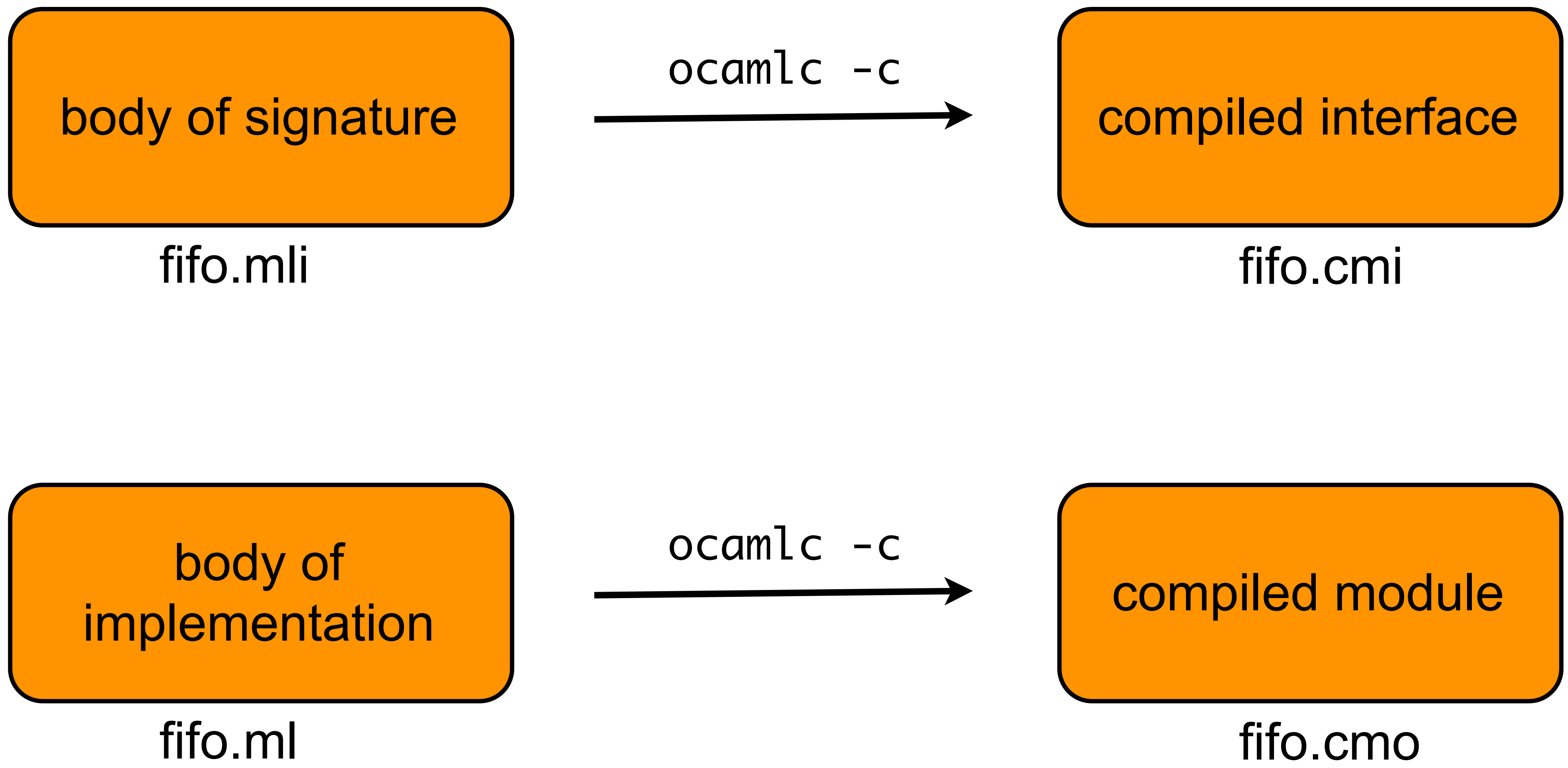
```
type comparison = Less | Equal | Greater;;

module type ORDERED_TYPE =
   sig
     type t
     val compare: t -> t -> comparison
   end;;

module Set =
   functor (Elt: ORDERED_TYPE) ->
     struct
       type element = Elt.t
       type set = element list
       let empty = [ ]
       let rec add x s =
         match s with
           [ ] -> [x]
         | hd::tl ->
            match Elt.compare x hd with
              Equal   -> s          (* x is already in s *)
            | Less    -> x :: s     (* x is smaller than all elements of s *)
            | Greater -> hd :: add x tl
       let rec member x s =
          ... WRITE IT! ...
     end;;
```

- functors are functions from modules to modules

Modules and Separate Compilation



`ocamlc fifo.cmo module2.cmo ... main.ml` produces `a.out` file

AST and calculi on terms

- structural induction == recursion + pattern-matching

```
type term = Var of string | Const of int | Plus of term * term | Minus of term * term
         | Mult of term * term | Ifz of term * term * term;;

type envt = (string * term) list;;

let rec eval t e = match t with
  | Var(x) -> List.assoc x e
  | Const(n) -> n
  | Plus(t1, t2) -> (eval t1 e) + (eval t2 e)
  | Minus(t1, t2) -> (eval t1 e) - (eval t2 e)
  | Mult(t1, t2) -> (eval t1 e) * (eval t2 e)
  | Ifz(p, t1, t2) -> if (eval p e = 0) then eval t1 e else eval t2 e ;;

let t = Minus(Plus(Mult( Const(3), Var ("x")), Var("y")),
              Mult(Const(2), Var("z")));;

let e = [("x", 45); ("y", 22); ("z", 17)] ;;
```

AST and calculi on terms

- structural induction == recursion + pattern-matching

```
let rec derivative t x = match t with
  | Var(y) -> if x = y then Const (1) else Const (0)
  | Const(n) -> Const (0)
  | Plus(t1, t2) -> Plus (derivative t1 x, derivative t2 x)
  | Minus(t1, t2) -> Minus (derivative t1 x, derivative t2 x)
  | Mult(t1, t2) -> Plus (Mult (t1, derivative t2 x), Mult (derivative t1 x, t2))
  | Ifz(p, t1, t2) -> Ifz(p, derivative t1 x, derivative t2 x) ;;
```

- evaluate expression below. What's missing ?

```
simplify (derivative t "x")
```

- beautiful program. How to make it more efficient ?

AST and calculi on terms

- structural induction == recursion + pattern-matching

```
let contract t = match t with
  | Plus (Const(0), v) -> v
  | Plus (u, Const(0)) -> u
  | Mult (Const(0), v) -> Const(0)
  | Mult (u, Const(0)) -> Const(0)
  | Mult (Const(1), v) -> v
  | Mult (u, Const(1)) -> u
  | Ifz (Const(0), u, v) -> u
  | Ifz (Const(1), u, v) -> v
  | _ -> t ;;

let rec simplify t = match t with
  | Plus(u,v) -> contract (Plus(simplify u, simplify v))
  | Minus(u,v) -> contract (Minus(simplify u, simplify v))
  | Mult(u,v) -> contract (Mult(simplify u, simplify v))
  | Ifz(p,u,v) -> contract (Ifz(simplify p, simplify u, simplify v))
  | Var(y) -> t
  | Const(n) -> t ;;
```

Modules vs Objects

- procedural programming : Fortran, Algol, Pascal, C, ML, Haskell, all functional prog.
- data-oriented programming: OOP, Simula, Smalltalk, C++, Java, C#, Eiffel, Python, Scala

	data modification	code modification
procedural prog.	global	local
OO prog.	local	global

- OOP promotes incremental programming with **inheritance** «modules are open»
- Elegant for graphics or remote computing
- but **dangerous** ! and often **hard to read**

Objects in Ocaml

- compromise between multiple inheritance and type inference
- complex theory and implementation

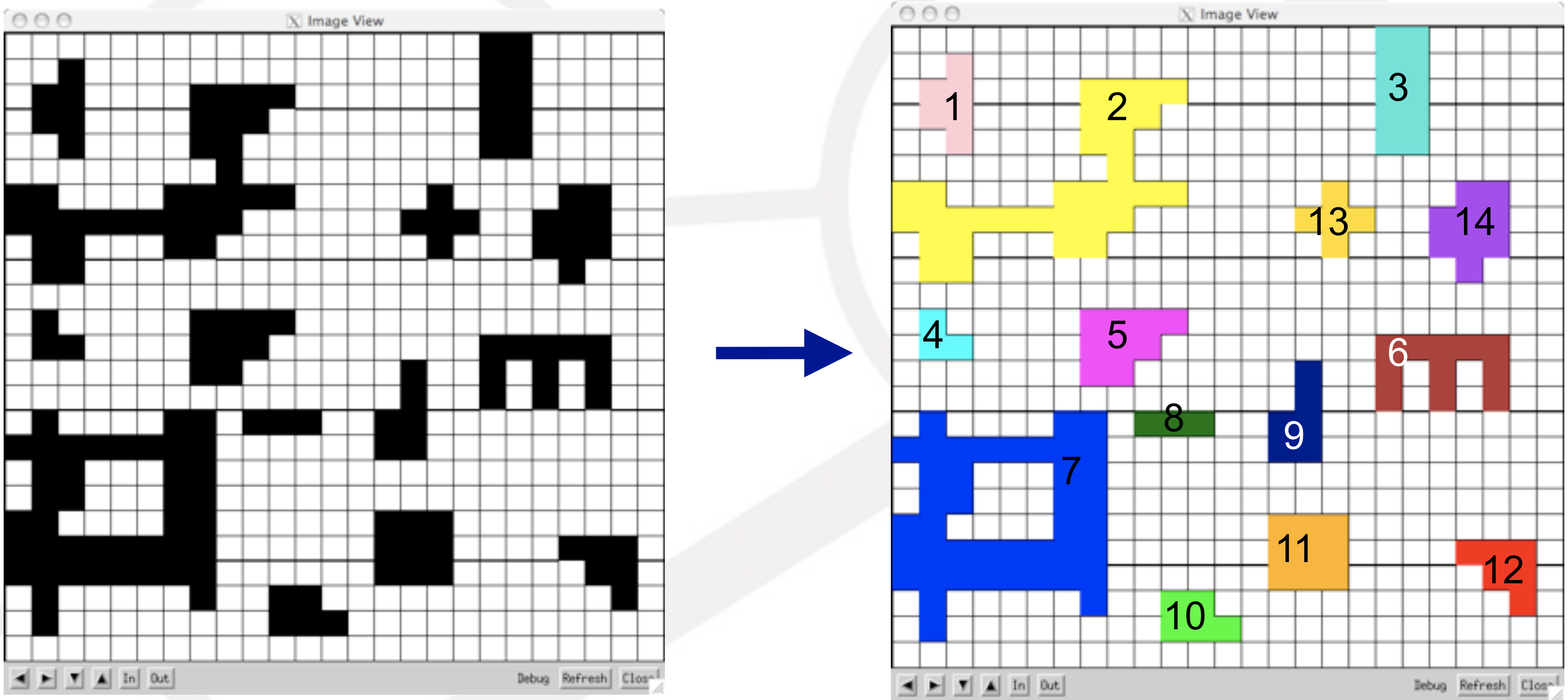
```
# class point =
   object
     val mutable x = 0
     val mutable y = 0
     method get_x = x
     method get_y = y
     method move dx dy  = x <- x + dx; y <- y + dy
   end;;
             class point :
  object
    val mutable x : int
    val mutable y : int
    method get_x : int
    method get_y : int
    method move : int -> int -> unit
  end
# let p = new point ;;
val p : point = <obj>
# p#get_x;;
- : int = 0
# p#move 10 20;;
- : unit = ()
# p#get_y;;
- : int = 20
```

Combien d'objets dans une image?

Jean-Jacques Lévy
INRIA

CENTRE DE RECHERCHE COMMUN
INRIA
MICROSOFT RESEARCH

Labeling



16 objects in this picture

Algorithm

1) first pass

- scan pixels left-to-right, top-to-bottom giving a new object id each time a new object is met

2) second pass

- generate equivalences between ids due to new adjacent relations met during scan of pixels.

3) third pass

- compute the number of equivalence classes

Complexity:

- scan twice full image (linear cost)
- try to efficiently manage equivalence classes (Union-Find by Tarjan)

Program structure

- `int array` of colored pixels as input
- `int array` of object ids (end of pass 1)
- `int array` of final object ids (end of pass 2)
- the `image` dumped on screen by Ocaml graphics primitives
- an colorful array of great number of colors to pick in to show final objects with distinct colors [take the one given on next slide]

```
(* colors *)

let maroon = 0x800000 ;;
let darkred = 0x8B0000 ;;
let red = 0xFF0000 ;;
let lightpink = 0xFFB6C1 ;;
let crimson = 0xDC143C ;;
let palevioletred = 0xDB7093 ;;
let hotpink = 0xFF69B4 ;;
let deeppink = 0xFF1493 ;;
let mediumvioletred = 0xC71585 ;;
let purple = 0x800080 ;;
let darkmagenta = 0x8B008B ;;
let orchid = 0xDA70D6 ;;
let thistle = 0xD8BFD8 ;;
let plum = 0xDDA0DD ;;
let violet = 0xEE82EE ;;
let magenta = 0xFF00FF ;;
let mediumorchid = 0xBA55D3 ;;
let darkviolet = 0x9400D3 ;;
let blue = 0x0000FF ;;
let navy = 0x000080 ;;
let royalblue = 0x4169E1 ;;
let skyblue = 0x87CEEB ;;

let teal = 0x008080 ;;
let cyan = 0x00FFFF ;;
let paleturquoise = 0xAFEEEE ;;
let turquoise = 0x40E0D0 ;;
let darkslategray = 0x2F4F4F ;;
let mediumspringgreen = 0x00FA9A ;;
let mediumaquamarine = 0x66CDAA ;;
let springgreen = 0x00FF7F ;;
let green = 0x008000 ;;
let olivedrab = 0x6B8E23 ;;
let darkkhaki = 0xBDB76B ;;
let gold = 0xFFD700 ;;
let darkgoldenrod = 0xB8860B ;;
let darkorange = 0xFF8C00 ;;
let chocolate = 0xD2691E ;;
let sienna = 0xA0522D ;;
let lightsalmon = 0xFFA07A ;;
let darksalmon = 0xE9967A ;;
let mistyrose = 0xFFE4E1 ;;
let orangered = 0xFF4500 ;;
let pink = 0xFFC0CB ;;
let lightcoral = 0xF08080 ;;
let brown = 0xA52A2A ;;
let firebrick = 0xB22222 ;;

let colors = [|
  maroon ;
  royalblue ;
  darksalmon ;
  darkred ;
  cyan ;
  darkkhaki ;
  orchid ;
  orangered ;
  purple ;
  hotpink ;
  turquoise ;
  chocolate ;
  red ;
  navy ;
  sienna ;
  mediumorchid ;
  lightpink ;
  springgreen ;
  violet ;
  mediumaquamarine ;
  firebrick ;
  olivedrab ;
  crimson ;
  skyblue ;
  lightcoral ;
  paleturquoise ;
  deeppink ;
  green ;
  darkorange ;
  blue ;
  darkgoldenrod ;
  darkmagenta ;
  teal ;
  mistyrose ;
  darkslategray ;
  magenta ;
  lightsalmon ;
  darkviolet ;
  mediumspringgreen ;
  pink ;
  plum ;
  gold ;
  brown ;
  mediumvioletred ;
  thistle
|] ;;
```