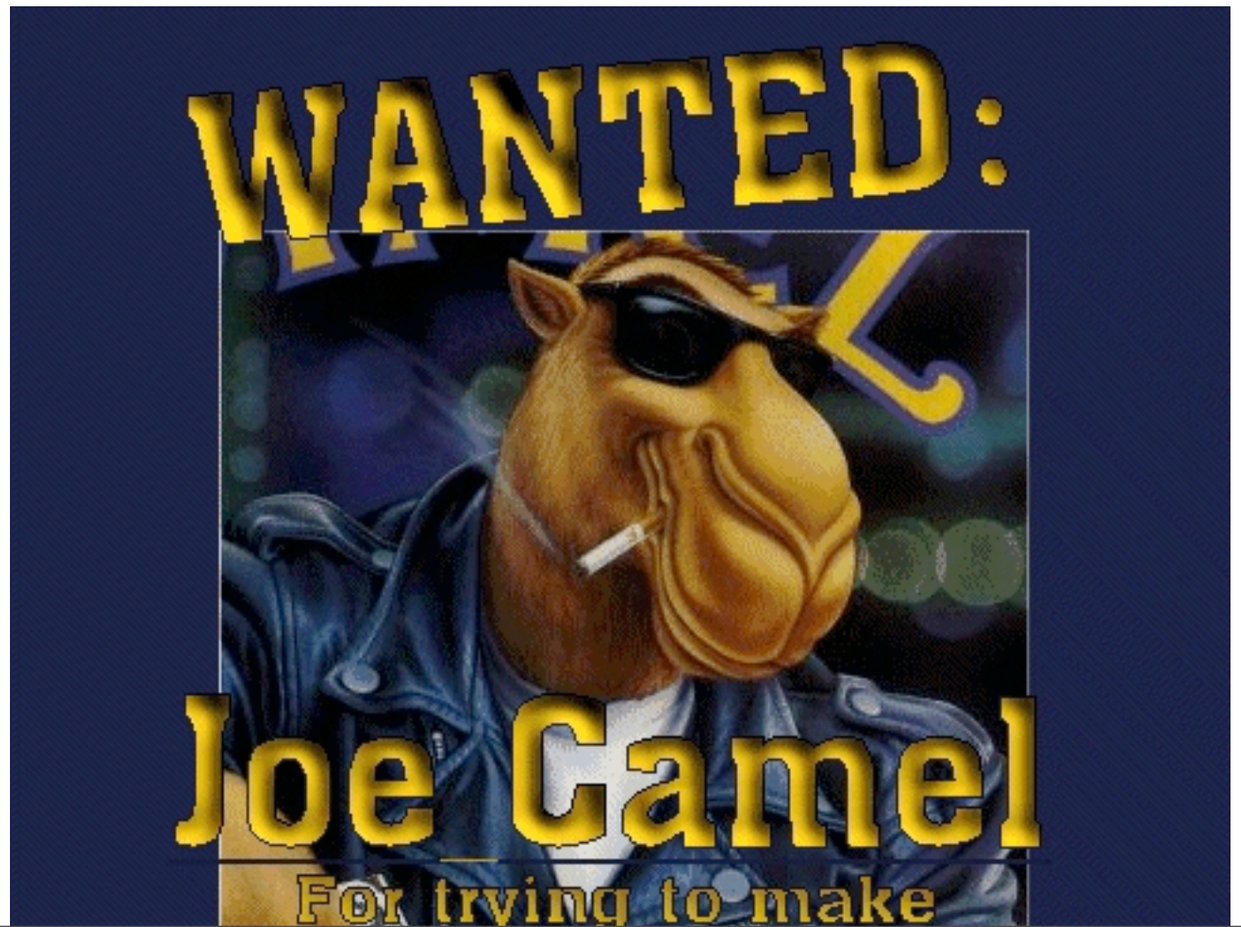
J-O-Caml (3)

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

Qinghua, November 26



Plan of this class

- records
- references and mutable data
- input/output
- exceptions
- a tour in library
- modules and interfaces
- labeling algorithm

Exercices

- Conway sequences - solution 1

```
# let print_list x =
  List.iter (function a -> Printf.printf "%d " a) x ; Printf.printf "\n" ;;
  val print_list : int list -> unit = <fun>
# let rec conway x = match x with
  | [ ] -> [ ]
  | a :: x' -> let y = conway x' in match y with
      | [ ] -> [1; a]
      | n :: b :: y' -> if a = b then (n+1) :: b :: y' else 1 :: a :: y
      | _ -> failwith "Impossible" ;;
  val conway : int list -> int list = <fun>
# let rec conways x n =
    print_list x; if n > 0 then conways (conway x) (n-1) ;;
  val conways : int list -> int -> unit = <fun>
```

- Conway sequences - solution 2 (with less many conses) ?

Zero-ary functions

- functions are monadic in Caml
- type constructors (which are not functions) have arity (maybe 0)

```
# let x = () and f () = 1 ;;
val x : unit = ()
val f : unit -> int = <fun>
# f x ;;
- : int = 1
# type color = Red | Yellow ;;
type color = Red | Yellow
# Red ;;
- : color = Red
# Red () ;;
Characters 0-6:
  Red () ;;
  ^^^^^^
Error: The constructor Red expects 0 argument(s),
       but is applied here to 1 argument(s)
# type tree = Empty | Node of tree * int * tree ;;
type tree = Empty | Node of tree * int * tree
```

Records

- type ``record'' needs be declared

```
# type course = { instructor : string; mutable students : string list; };;

# let jocaml = {instructor = "JJL"; students = ["william"; "bill"] };;

# jocaml.students <- "lin" :: jocaml.students;;

# jocaml;;

# let student_list = [
  "Chen Danning";
  "Gao Jianhua" ;
  "Hong Ali" ;
  "Ji Xu" ;
  "Jiang Huixiang" ];;

# jocaml.students <- student_list;;

# jocaml;;
```

Records

- type ``record'' needs be declared

```
# type course = { instructor : string; mutable students : string list; };;
type course = { instructor : string; mutable students : string list; }
# let jocaml = {instructor = "JJL"; students = ["william"; "bill"] };;
val jocaml : course = {instructor = "JJL"; students = ["william"; "bill"]}
# jocaml.students <- "lin" :: jocaml.students;;
- : unit = ()
# jocaml;;
- : course = {instructor = "JJL"; students = ["lin"; "william"; "bill"]}
# let student_list = [
  "Chen Danning";
  "Gao Jianhua" ;
  "Hong Ali" ;
  "Ji Xu" ;
  "Jiang Huixiang" ];;
val student_list : string list =
  ["Chen Danning"; "Gao Jianhua"; "Hong Ali"; "Ji Xu"; "Jiang Huixiang"]
# jocaml.students <- student_list;;
- : unit = ()
# jocaml;;
- : course =
{instructor = "JJL";
 students =
  ["Chen Danning"; "Gao Jianhua"; "Hong Ali"; "Ji Xu"; "Jiang Huixiang"]}
```

Mutable fields in records

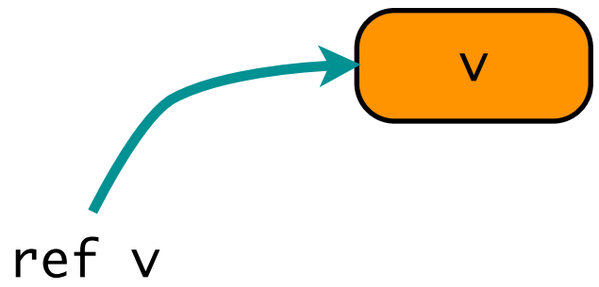
- several fields may be declared **mutable** in records (`students` in previous example)
- until now, all variables were **constant**
- important information for garbage collector, parallel evaluator, caches, etc
- constant values are less error-prone than mutable values, especially with sharing, concurrency, etc.
- in C, C++, Java, etc, variables are mutable by default
- in ML, it's the opposite
- Keeping variables constant is the basis of **Functional Programming** (**no side-effects**)
- In Haskell, mutable world (monads) and constant world (usual expressions) are distinct.

References

- `ref v` is L-value of the mutable value `v` (a pointer address!)
- `!x` dereferences x and produces `v`
- `:=` modifies the value of a reference

  (Beware: `:=` for references; `<-` for arrays and strings!!)
- a reference is equivalent to a record with a single mutable field `contents`

```
# let oneEuro = ref 10.0 ;;
val oneEuro : float ref = {contents = 10.}
# !oneEuro ;;
- : float = 10.
# oneEuro := 10.154 ;;
- : unit = ()
# !oneEuro ;;
- : float = 10.154
```



Imperative programming

- with references, records, strings and arrays, one can use the imperative style of C, C++, Java, etc.
- however dereferencing of references must be explicit (no R-values)

```
# let main n x =
    let y = ref x in
    for i = 1 to n do
      print_list !y;
      y := conway !y
    done;;
val main : int -> int list -> unit = <fun>
```

Imperative programming

- sorting arrays (a la Sedgewick)

```
# let insertionSort a =
 let n = Array.length a in
 let j = ref 0 in
 for i = 1 to n - 1 do
   let v = ref a.(i) in
       begin
       j := i;
       while !j > 0 && a.(!j - 1) > !v do
         a.(!j) <- a.(!j - 1);
         decr j
       done;
       a.(!j) <- !v;
       end
 done;;
val insertionSort : 'a array -> unit = <fun>
```

Exceptions

- There are several built-in exceptions
- `Failure`, `Division_by_zero`, `Invalid_argument`, etc
- but exceptions may also be declared by:
- `raise` and `try ... with ...` handle exceptions with pattern-matching

```
try e with
| exception_1 -> e_1
| exception_2 -> e_2
...
| exception_n -> e_n
```

Input/Output

```
open_in : string -> in_channel
open_out : string -> out_channel
stdin : in_channel
stdout : out_channel
stderr : out_channel

input_char : in_channel -> char
input_line : in_channel -> string
input : in_channel -> string -> int -> int -> int
output_char : out_channel -> char -> unit
output_string : out_channel -> string -> unit
output : out_channel -> string -> int -> int -> unit

flush : out_channel -> unit
close_in : in_channel -> unit
close_out : out_channel -> unit
```

```
print_char : char -> unit
print_string : string -> unit
print_int : int -> unit
print_float : float -> unit
print_newline : unit -> unit

read_line : unit -> string
read_int : unit -> int
read_float : unit -> float

Printf.printf : ('a, out_channel, unit) format -> 'a
Scanf.scanf : ('a, 'b, 'c, 'd) Scanf.scanner
```

Input/Output

```
open Printf;;

let inWord = true and notInWord = false;;

type resultat = { mutable chars: int; mutable words: int; mutable lines: int } ;;

let file =  {chars = 0 ; words = 0 ; lines = 0};;
let total = {chars = 0 ; words = 0 ; lines = 0};;

let reset_count () = file.chars <- 0; file.words <- 0; file.lines <- 0 ;;

let cumulate () =
  total.chars <- total.chars + file.chars;
  total.words <- total.words + file.words;
  total.lines <- total.lines + file.lines;;

let rec counter f in_word =
  let c = input_char f in
  file.chars <- file.chars + 1;
  match c with
    ' ' | '\t' | '\n' ->
      if in_word then
        file.words <- file.words + 1;
      if c = '\n' then
        file.lines <- file.lines + 1;
      counter f notInWord
  | _ ->
      counter f inWord;;
```

Input/Output

```
let word_count_ch f =
  reset_count ();
  try counter f notInWord with
    End_of_file -> begin
      cumulate ();
      close_in f
    end;;

let output_results filename =
  printf " %9d %9d %9d %s\n"
         file.lines file.words file.chars
         filename;;

let ouput_total () =
  printf " %9d %9d %9d %s\n"
         total.lines total.words total.chars
         "total";;

let word_count_file filename =
  try
    let f = open_in filename in
    word_count_ch f;
    output_results (filename)
  with Sys_error s -> begin
    printf "%s\n" s; exit 2
    end;;
```

```
let main () =
  let nargs = Array.length (Sys.argv) - 1 in
  for i = 1 to nargs do
    word_count_file Sys.argv.(i)
  done;
  if nargs > 1 then ouput_total ();
  exit 0;;

main();;
```

Modules

- modules group functions of same nature
- **qualified notation** `Array.make, List.length` as in Java, Modula, etc
- they can be **opened** as in `open Printf`
- module `Pervasives` always open
- fetch modules in documentation at

  `caml.inria.fr/pub/docs/manual-ocaml`
- module `Graphics` is a portable **graphics** library (needs `graphics.cma` to be compiled as first argument of the `ocamlc` command)
- module names (`List`) start with uppercase letter
- and correspond to interfaces (`list.cmi`) starting with lowercase letter.

Graphics

```
open Graphics;;

let main() =
  open_graph "";

  set_line_width 1 ;
  set_color red ;
  fill_rect 10 10 100 200 ;;
```


ocamlc graphics.cma g1.ml

a.out

Graphics

- elementary functions `moveto, lineto, draw_rect, fill_rect, ...`
- type `color` is `int`
- **images** are internal representation of bitmaps
- a matrix of colors can be made into an image `make_image`
- an image can be displayes `dump_image`

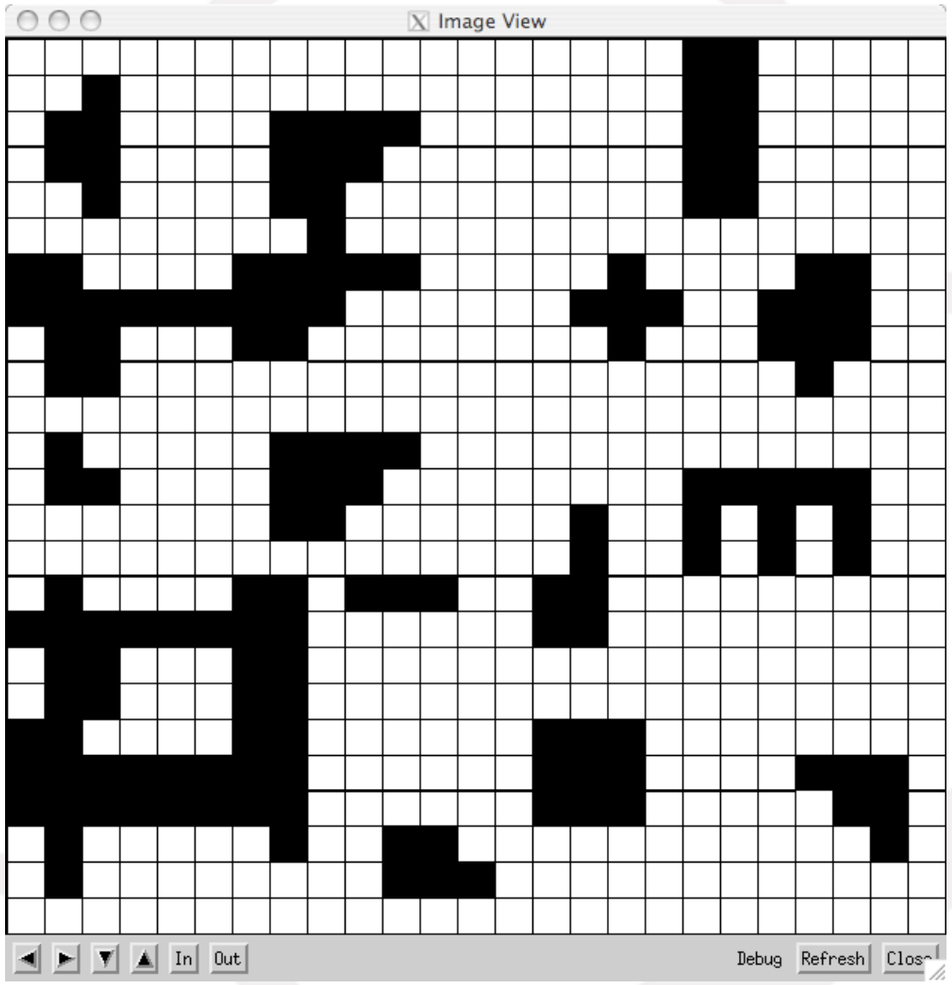
Combien d'objets dans une image?

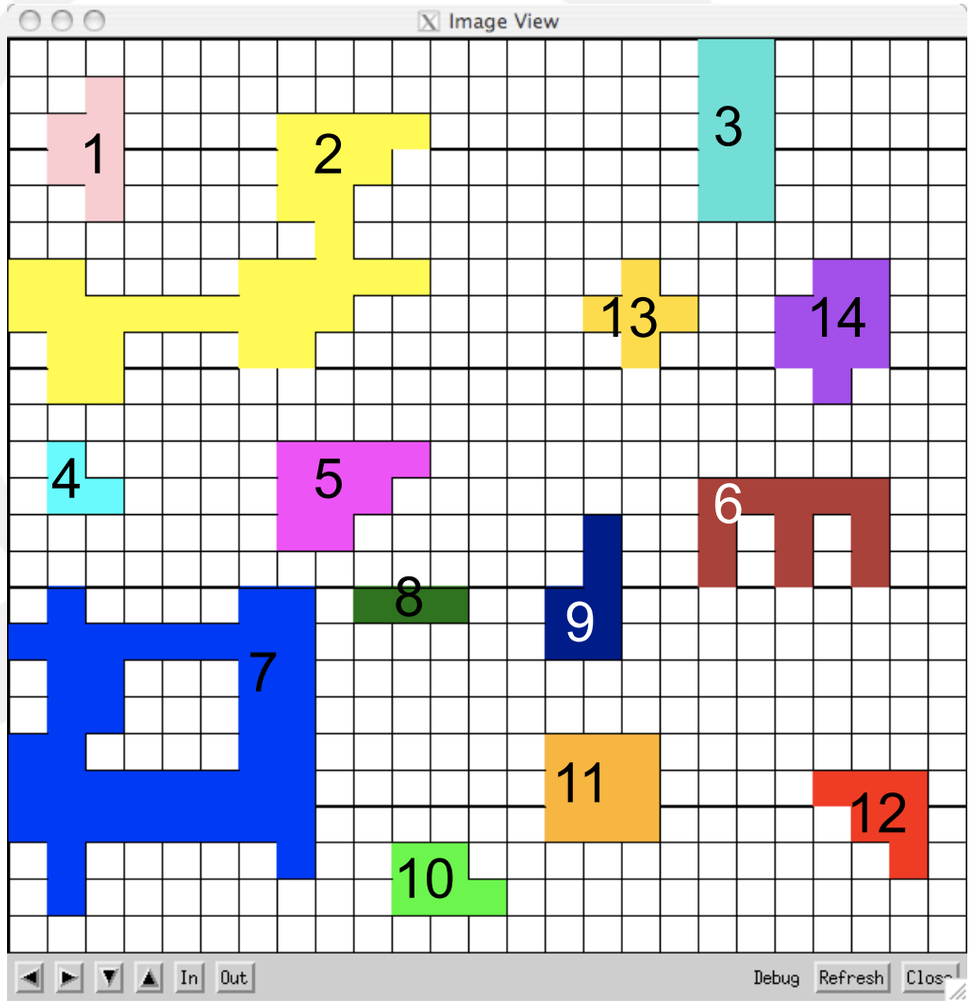
Jean-Jacques Lévy
INRIA

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)