École Polytechnique

INF549

A Short Introduction to OCaml

Jean-Christophe Filliâtre

September 11, 2018

overview

lecture
- Jean-Christophe Filliâtre

labs
- Stéphane Lengrand
- Monday 17 and Tuesday 18, 9h–12h

web site for this course

http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Christophe.Filliatre/INF549/

questions ⇒ Jean-Christophe.Filliatre@lri.fr

OCaml

OCaml is a general-purpose, strongly typed programming language

successor of Caml Light (itself successor of Caml),
part of the ML family (SML, F#, etc.)

designed and implemented at Inria Rocquencourt by Xavier Leroy and others

Some applications: symbolic computation and languages (IBM, Intel, Dassault Systèmes), static analysis (Microsoft, ENS), file synchronization (Unison), peer-to-peer (MLDonkey), finance (LexiFi, Jane Street Capital), teaching

first steps with OCaml

the first program

hello.ml

```
print_string "hello world!\n"
```

compiling

```
% ocamlopt -o hello hello.ml
```

executing

```
% ./hello
hello world!
```

the first program

hello.ml

```
print_string "hello world!\n"
```

compiling

```
% ocamlopt -o hello hello.ml
```

executing

```
% ./hello
hello world!
```

the first program

hello.ml

```
print_string "hello world!\n"
```

compiling

```
% ocamlopt -o hello hello.ml
```

executing

```
% ./hello
hello world!
```

declarations

program = sequence of declarations and expressions to evaluate

```
let x = 1 + 2;;
print_int x;;
let y = x * x;;
print_int y;;
```

declarations

program = sequence of declarations and expressions to evaluate

```
let x = 1 + 2;;
print_int x;;
let y = x * x;;
print_int y;;
```

variable

`let` x = e introduces a global variable

differences wrt usual notion of variable:

1. necessarily initialized
2. type not declared but inferred
3. cannot be assigned

Java	OCaml
`final int x = 42;`	`let x = 42`

references

a variable to be assigned is called a reference

it is introduced with `ref`

```
let x = ref 1;;
print_int !x;;
x := !x + 1;;
print_int !x;;
```

references

a variable to be assigned is called a reference

it is introduced with `ref`

```
let x = ref 1;;
print_int !x;;
x := !x + 1;;
print_int !x;;
```

expressions and statements

no distinction between expression/statement in the syntax : only expressions

usual constructs:

- conditional

```
if i = 1 then 2 else 3
```

- for loop

```
for i = 1 to 10 do x := !x + i done
```

- sequence

```
x := 1; 2 * !x
```

expressions and statements

no distinction between expression/statement in the syntax : only expressions

usual constructs:

- conditional

```
if i = 1 then 2 else 3
```

- for loop

```
for i = 1 to 10 do x := !x + i done
```

- sequence

```
x := 1; 2 * !x
```

expressions and statements

no distinction between expression/statement in the syntax : only expressions

usual constructs:

- conditional

```
if i = 1 then 2 else 3
```

- `for` loop

```
for i = 1 to 10 do x := !x + i done
```

- sequence

```
x := 1; 2 * !x
```

expressions and statements

no distinction between expression/statement in the syntax : only expressions

usual constructs:

- conditional

```
if i = 1 then 2 else 3
```

- `for` loop

```
for i = 1 to 10 do x := !x + i done
```

- sequence

```
x := 1; 2 * !x
```

unit type

expressions with no meaningful value (assignment, loop, ...) have type `unit`

this type has a single value, written `()`

it is the type given to the `else` branch when it is omitted

correct:

```
if !x > 0 then x := 0
```

incorrect:

```
2 + (if !x > 0 then 1)
```

unit type

expressions with no meaningful value (assignment, loop, ...) have type `unit`

this type has a single value, written `()`

it is the type given to the `else` branch when it is omitted

correct:

```
if !x > 0 then x := 0
```

incorrect:

```
2 + (if !x > 0 then 1)
```

unit type

expressions with no meaningful value (assignment, loop, ...) have type `unit`
this type has a single value, written `()`

it is the type given to the `else` branch when it is omitted

correct:

```
if !x > 0 then x := 0
```

incorrect:

```
2 + (if !x > 0 then 1)
```

local variables

in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...
}
```

in OCaml, a local variable is introduced with `let in`:

```
  let x = 10 in x * x
```

as for a global variable:

- necessarily initialized
- type inferred
- immutable
- but scope limited to the expression following `in`

local variables

in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...
}
```

in OCaml, a local variable is introduced with `let in`:

```
  let x = 10 in x * x
```

as for a global variable:

- necessarily initialized
- type inferred
- immutable
- but scope limited to the expression following `in`

local variables

in C or Java, the scope of a local variable extends to the bloc:

```
{
  int x = 1;
  ...
}
```

in OCaml, a local variable is introduced with `let in`:

```
  let x = 10 in x * x
```

as for a global variable:

- necessarily initialized
- type inferred
- immutable
- but scope limited to the expression following `in`

let in = expression

let x = e1 in e2 is an expression

its type and value are those of e2,

in an environment where x has the type and value of e1

example

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

let in = expression

let x = e1 in e2 is an expression
its type and value are those of e2,
in an environment where x has the type and value of e1

example

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

let in = expression

let x = e1 in e2 is an expression
its type and value are those of e2,
in an environment where x has the type and value of e1

example

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

parallel

Java	OCaml

`{ int x = 1;`
`  x = x + 1;`
`  int y = x * x;`
`  System.out.print(y); }`
	`let x = ref 1 in`
	`x := !x + 1;`
	`let y = !x * !x in`
	`print_int y`

recap

- program = sequence of expressions and declarations
- variables introduced with `let` and immutable
- no distinction expression / statement

interactive loop

interactive version of the compiler

```
% ocaml
        OCaml version 4.02.3

# let x = 1 in x + 2;;
```

```
- : int = 3
```

```
# let y = 1 + 2;;
```

```
val y : int = 3
```

```
# y * y;;
```

```
- : int = 9
```

functions

syntax

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- body = expression (no `return`)
- type is inferred (types of argument `x` and result)

```
# f 4;;
```

```
- : int = 16
```

syntax

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- body = expression (no `return`)
- type is inferred (types of argument `x` and result)

```
# f 4;;
```

```
- : int = 16
```

syntax

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- body = expression (no `return`)
- type is inferred (types of argument `x` and result)

```
# f 4;;
```

```
- : int = 16
```

syntax

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- body = expression (no `return`)
- type is inferred (types of argument `x` and result)

```
# f 4;;
```

```
- : int = 16
```

parallel

Java	OCaml

`static int f(int x) {`
`   return x * x;`
`}`
	`let f x =`
	`   x * x`

procedure

a procedure = a function whose result type is unit

example

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

procedure

a procedure = a function whose result type is `unit`

example

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

procedure

a procedure = a function whose result type is unit

example

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

procedure

a procedure = a function whose result type is unit

example

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

function without arguments

takes an argument of type `unit`

example

```
# let reset () = x := 0;;
```

```
val reset : unit -> unit = <fun>
```

```
# reset ();;
```

function without arguments

takes an argument of type unit

example

```
# let reset () = x := 0;;
```

```
val reset : unit -> unit = <fun>
```

```
# reset ();;
```

function with several arguments

```
# let f x y z = if x > 0 then y + x else z - x;;
```

```
val f : int -> int -> int -> int = <fun>
```

```
# f 1 2 3;;
```

```
- : int = 3
```

function with several arguments

```
# let f x y z = if x > 0 then y + x else z - x;;
```

```
val f : int -> int -> int -> int = <fun>
```

```
# f 1 2 3;;
```

```
- : int = 3
```

local function

function local to an expression

```
# let sqr x = x * x in sqr 3 + sqr 4 = sqr 5;;
```

```
- : bool = true
```

function local to another function

```
# let pythagorean x y z =
    let sqr n = n * n in
    sqr x + sqr y = sqr z;;
```

```
val pythagorean : int -> int -> int -> bool = <fun>
```

local function

function local to an expression

```
# let sqr x = x * x in sqr 3 + sqr 4 = sqr 5;;
```

```
- : bool = true
```

function local to another function

```
# let pythagorean x y z =
    let sqr n = n * n in
    sqr x + sqr y = sqr z;;
```

```
val pythagorean : int -> int -> int -> bool = <fun>
```

function as first-class citizen

function = yet another expression, introduced with `fun`

```
# fun x -> x+1
```

```
- : int -> int = <fun>
```

```
# (fun x -> x+1) 3;;
```

```
- : int = 4
```

internally

```
let f x = x+1;;
```

is identical to

```
let f = fun x -> x+1;;
```

function as first-class citizen

function = yet another expression, introduced with fun

```
# fun x -> x+1
```

```
- : int -> int = <fun>
```

```
# (fun x -> x+1) 3;;
```

```
- : int = 4
```

internally

```
let f x = x+1;;
```

is identical to

```
let f = fun x -> x+1;;
```

partial application

```
fun x y -> x*x + y*y
```

is the same as

```
fun x -> fun y -> x*x + y*y
```

one can apply a function partially

example

```
# let f x y = x*x + y*y;;
```

```
val f : int -> int -> int = <fun>
```

```
# let g = f 3;;
```

```
val g : int -> int = <fun>
```

```
# g 4;;
```

```
- : int = 25
```

partial application

```
fun x y -> x*x + y*y
```

is the same as

```
fun x -> fun y -> x*x + y*y
```

one can apply a function partially

example

```
# let f x y = x*x + y*y;;
```

```
val f : int -> int -> int = <fun>
```

```
# let g = f 3;;
```

```
val g : int -> int = <fun>
```

```
# g 4;;
```

```
- : int = 25
```

partial application

a partial application is a way to return a function

but one can also return a function as the result of a computation

```
# let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

```
val f : int -> int -> int = <fun>
```

a partial application of `f` computes `x*x` only once

partial application: example

```
# let count_from n =
    let r = ref (n-1) in fun () -> incr r; !r;;
```

```
val count_from : int -> unit -> int = <fun>
```

```
# let c = count_from 0;;
```

```
val c : unit -> int = <fun>
```

```
# c ();;
```

```
- : int = 0
```

```
# c ();;
```

```
- : int = 1
```

higher-order functions

a function may take functions as arguments

```
# let integral f =
    let n = 100 in
    let s = ref 0.0 in
    for i = 0 to n-1 do
      let x = float i /. float n in s := !s +. f x
    done;
    !s /. float n
```

```
# integral sin;;
```

```
- : float = 0.455486508387318301
```

```
# integral (fun x -> x*.x);;
```

```
- : float = 0.32835
```

iteration

in Java, one iterates over a collection with a cursor

```
for (Elt x: s) {
    ... do something with x ...
}
```

in OCaml, we typically write

```
iter (fun x -> ... do something with x ...) s
```

where `iter` is a function provided with the data structure, with type

```
val iter: (elt -> unit) -> set -> unit
```

example

```
iter (fun x -> Printf.printf "%s\n" x) s
```

iteration

in Java, one iterates over a collection with a cursor

```
for (Elt x: s) {
    ... do something with x ...
}
```

in OCaml, we typically write

```
iter (fun x -> ... do something with x ...) s
```

where `iter` is a function provided with the data structure, with type

```
val iter: (elt -> unit) -> set -> unit
```

example

```
iter (fun x -> Printf.printf "%s\n" x) s
```

difference wrt to function pointers

"in C one can pass and return function pointers"

but OCaml functions are more than function pointers

```
let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

the value of `x2` is captured in a **closure**

note: there are closures in Java ($\geq$ 8) too

```
s.forEach(x -> { System.out.println(x); });
```

difference wrt to function pointers

"in C one can pass and return function pointers"

but OCaml functions are more than function pointers

```
let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

the value of `x2` is captured in a **closure**

note: there are closures in Java (≥ 8) too

```
s.forEach(x -> { System.out.println(x); });
```

recursive functions

in OCaml, it is idiomatic to use recursive functions, for

- a function call is cheap
- tail calls are optimized

example:

```
let zero f =
  let rec lookup i = if f i = 0 then i else lookup (i+1) in
  lookup 0
```

recursive code $\Rightarrow$ clearer, simpler to justify

recursive functions

in OCaml, it is idiomatic to use recursive functions, for

- a function call is cheap
- tail calls are optimized

example:

```
let zero f =
  let rec lookup i = if f i = 0 then i else lookup (i+1) in
  lookup 0
```

recursive code $\Rightarrow$ clearer, simpler to justify

polymorphism

```
# let f x = x;;
```

```
val f : 'a -> 'a = <fun>
```

```
# f 3;;
```

```
- : int = 3
```

```
# f true;;
```

```
- : bool = true
```

```
# f print_int;;
```

```
- : int -> unit = <fun>
```

```
# f print_int 1;;
```

```
1- : unit = ()
```

polymorphism

```
# let f x = x;;
```

```
val f : 'a -> 'a = <fun>
```

```
# f 3;;
```

```
- : int = 3
```

```
# f true;;
```

```
- : bool = true
```

```
# f print_int;;
```

```
- : int -> unit = <fun>
```

```
# f print_int 1;;
```

```
1- : unit = ()
```

polymorphism

OCaml always infers the most general type

example:

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

polymorphism

OCaml always infers the most general type

example:

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

recap

- functions = first-class values: local, anonymous, arguments of other functions, etc.
- partially applied
- polymorphic
- function call is cheap

memory allocation

GC

memory allocation handled by a garbage collector (GC)

benefits:

- unused memory is reclaimed automatically
- efficient allocation

⇒ forget about "dynamic allocation is expensive"
... but keep worrying about complexity!

GC

memory allocation handled by a garbage collector (GC)

benefits:

- unused memory is reclaimed automatically
- efficient allocation

⇒ forget about "dynamic allocation is expensive"
... but keep worrying about complexity!

arrays

```
# let a = Array.make 10 0;;
```

```
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

necessarily initialized

```
# let a = [| 1; 2; 3; 4 |];;
```

```
# a.(1);;
```

```
- : int = 2
```

```
# a.(1) <- 5;;
```

```
- : unit = ()
```

```
# a;;
```

```
- : int array = [|1; 5; 3; 4|]
```

arrays

```
# let a = Array.make 10 0;;
```

```
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

necessarily initialized

```
# let a = [| 1; 2; 3; 4 |];;
```

```
# a.(1);;
```

```
- : int = 2
```

```
# a.(1) <- 5;;
```

```
- : unit = ()
```

```
# a;;
```

```
- : int array = [|1; 5; 3; 4|]
```

parallel

Java	OCaml
`int[] a = new int[42];`	`let a = Array.make 42 0`
`a[17]`	`a.(17)`
`a[7] = 3;`	`a.(7) <- 3`
`a.length`	`Array.length a`

example: insertion sort

```
let insertion_sort a =
  let swap i j =
    let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
  in
  for i = 1 to Array.length a - 1 do
    (* insert element a[i] in a[0..i-1] *)
    let j = ref (i - 1) in
    while !j >= 0 && a.(!j) > a.(!j + 1) do
      swap !j (!j + 1); decr j
    done
  done
```

insertion sort

```
let insertion_sort a =
  let swap i j =
    let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
  in
  for i = 1 to Array.length a - 1 do
    (* insert element a[i] in a[0..i-1] *)
    let rec insert j =
      if j >= 0 && a.(j) > a.(j+1) then
      begin swap j (j+1); insert (j-1) end
    in
    insert (i-1)
  done
```

records

like in most programming languages

a record type is first declared

```
type complex = { re : float; im : float }
```

allocation and initialization are simultaneous:

```
# let x = { re = 1.0; im = -1.0 };;
```

```
val x : complex = {re = 1.; im = -1.}
```

```
# x.im;;
```

```
- : float = -1.
```

records

like in most programming languages
a record type is first declared

```
type complex = { re : float; im : float }
```

allocation and initialization are simultaneous:

```
# let x = { re = 1.0; im = -1.0 };;
```

```
val x : complex = {re = 1.; im = -1.}
```

```
# x.im;;
```

```
- : float = -1.
```

records

like in most programming languages
a record type is first declared

```
type complex = { re : float; im : float }
```

allocation and initialization are simultaneous:

```
# let x = { re = 1.0; im = -1.0 };;
```

```
val x : complex = {re = 1.; im = -1.}
```

```
# x.im;;
```

```
- : float = -1.
```

mutable fields

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

```
# let p =  { name = "Martin"; age = 23 };;
```

```
val p : person = {name = "Martin"; age = 23}
```

```
# p.age <- p.age + 1;;
```

```
- : unit = ()
```

```
# p.age;;
```

```
- : int = 24
```

mutable fields

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

```
# let p =  { name = "Martin"; age = 23 };;
```

```
val p : person = {name = "Martin"; age = 23}
```

```
# p.age <- p.age + 1;;
```

```
- : unit = ()
```

```
# p.age;;
```

```
- : int = 24
```

parallel

Java	OCaml

`class T {`
`   final int v; boolean b;`
`   T(int v, boolean b) {`
`      this.v = v; this.b = b;`
`   }`
`}`
	`type t = {`
	`   v: int;`
	`   mutable b: bool;`
	`}`

`T r = new T(42, true);`	`let r = { v = 42; b = true }`
`r.b = false;`	`r.b <- false`
`r.v`	`r.v`

references

a reference = a record of that predefined type

```
type 'a ref = { mutable contents : 'a }
```

`ref`, `!` and `:=` are syntactic sugar

only arrays and mutable fields can be mutated

references

a reference = a record of that predefined type

```
type 'a ref = { mutable contents : 'a }
```

`ref`, `!` and `:=` are syntactic sugar

only arrays and mutable fields can be mutated

references

a reference = a record of that predefined type

```
type 'a ref = { mutable contents : 'a }
```

`ref`, `!` and `:=` are syntactic sugar

only arrays and mutable fields can be mutated

tuples

usual notation

```
# (1,2,3);;
```

```
- : int * int * int = (1, 2, 3)
```

```
# let v = (1, true, "hello", 'a');;
```

```
val v : int * bool * string * char =
  (1, true, "hello", 'a')
```

access to components

```
# let (a,b,c,d) = v;;
```

```
val a : int = 1
val b : bool = true
val c : string = "hello"
val d : char = 'a'
```

tuples

usual notation

```
# (1,2,3);;
```

```
- : int * int * int = (1, 2, 3)
```

```
# let v = (1, true, "hello", 'a');;
```

```
val v : int * bool * string * char =
  (1, true, "hello", 'a')
```

access to components

```
# let (a,b,c,d) = v;;
```

```
val a : int = 1
val b : bool = true
val c : string = "hello"
val d : char = 'a'
```

tuples

useful to return several values

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;
```

```
val division : int -> int -> int * int = <fun>
```

function taking a tuple as argument

```
# let f (x,y) = x + y;;
```

```
val f : int * int -> int = <fun>
```

```
# f (1,2);;
```

```
- : int = 3
```

tuples

useful to return several values

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;
```

```
val division : int -> int -> int * int = <fun>
```

function taking a tuple as argument

```
# let f (x,y) = x + y;;
```

```
val f : int * int -> int = <fun>
```

```
# f (1,2);;
```

```
- : int = 3
```

lists

predefined type of lists, α `list`, immutable and homogeneous
built from the empty list `[]` and addition in front of a list `::`

```
# let l = 1 :: 2 :: 3 :: [];;
```

```
val l : int list = [1; 2; 3]
```

shorter syntax

```
# let l = [1; 2; 3];;
```

pattern matching

pattern matching = case analysis on a list

```
# let rec sum l =
    match l with
    | []     -> 0
    | x :: r -> x + sum r;;
```

```
val sum : int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

shorter notation for a function performing pattern matching on its argument

```
let rec sum = function
  | []     -> 0
  | x :: r -> x + sum r;;
```

pattern matching

pattern matching = case analysis on a list

```
# let rec sum l =
    match l with
    | []     -> 0
    | x :: r -> x + sum r;;
```

```
val sum : int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

shorter notation for a function performing pattern matching on its argument

```
let rec sum = function
  | []     -> 0
  | x :: r -> x + sum r;;
```

pattern matching

pattern matching = case analysis on a list

```
# let rec sum l =
    match l with
    | []     -> 0
    | x :: r -> x + sum r;;
```

```
val sum : int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

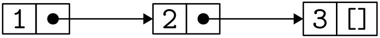
shorter notation for a function performing pattern matching on its argument

```
let rec sum = function
  | []     -> 0
  | x :: r -> x + sum r;;
```

representation in memory

OCaml lists = identical to lists in C or Java

the list `[1; 2; 3]` is represented as



algebraic data types

lists = particular case of algebraic data type

algebraic data type = union of several constructors

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

lists predefined as

```
type 'a list = [] | :: of 'a * 'a list
```

algebraic data types

lists = particular case of algebraic data type

algebraic data type = union of several constructors

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

lists predefined as

```
type 'a list = [] | :: of 'a * 'a list
```

algebraic data types

lists = particular case of algebraic data type

algebraic data type = union of several constructors

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

lists predefined as

```
type 'a list = [] | :: of 'a * 'a list
```

algebraic data types

lists = particular case of algebraic data type

algebraic data type = union of several constructors

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

lists predefined as

```
type 'a list = [] | :: of 'a * 'a list
```

pattern matching

pattern matching generalizes to algebraic data types

```
# let rec eval = function
    | True  -> true
    | False -> false
    | And (f1, f2) -> eval f1 && eval f2;;
```

```
val eval : fmla -> bool = <fun>
```

pattern matching

patterns can be nested:

```
let rec eval = function
  | True  -> true
  | False -> false
  | And (False, f2) -> false
  | And (f1, False) -> false
  | And (f1, f2)    -> eval f1 && eval f2;;
```

pattern matching

patterns can be omitted or grouped

```
let rec eval = function
  | True  -> true
  | False -> false
  | And (False, _) | And (_, False) -> false
  | And (f1, f2) -> eval f1 && eval f2;;
```

parallel

Java

```java
abstract class Fmla { }
class True extends Fmla { }
class False extends Fmla { }
class And extends Fmla {
  Fmla f1, f2; }

abstract class Fmla {
  abstract boolean eval(); }
class True { boolean eval() {
  return true; } }
class False { boolean eval() {
  return false; } }
class And { boolean eval() {
  return f1.eval()&&f2.eval();
} }
```

OCaml

```ocaml
type fmla =
| True
| False
| And of fmla * fmla


let rec eval = function
| True -> true

| False -> false

| And (f1, f2) ->
    eval f1 && eval f2
```

pattern matching

pattern matching is not limited to algebraic data types

```
let rec mult = function
  | []     -> 1
  | 0 :: _ -> 0
  | x :: l -> x * mult l
```

one may write `let pattern = expression` when there is a single pattern (as in `let (a,b,c,d) = v` for instance)

pattern matching

pattern matching is not limited to algebraic data types

```
let rec mult = function
  | []     -> 1
  | 0 :: _ -> 0
  | x :: l -> x * mult l
```

one may write `let pattern = expression` when there is a single pattern (as in `let (a,b,c,d) = v` for instance)

recap

- allocation is cheap
- memory is reclaimed automatically
- allocated values are necessarily initialized
- most values cannot be mutated
  (only arrays and mutable record fields can be)
- efficient representation of values
- pattern matching = case analysis over values

execution model

values

a value is

- either a primitive value (integer, floating point, Boolean, `[]`, etc.)
- or a pointer (to an array, a constructor such as `And`, etc.)

it fits on 64 bits

passing mode is **by value**

in particular, no value is ever copied

it is **exactly as in Java**

no `null` value

in OCaml, there is **no such thing as `null`**

in particular, any value is necessarily initialized

sometimes a pain, but it's worth the effort:

> *an expression of type τ whose evaluation terminates necessarily has a legal value of type τ*

this is known as strong typing

no such thing as `NullPointerException`
(neither `segmentation fault` as in C/C++)

comparison

equality written == is **physical equality**,
that is, equality of pointers or primitive values

```
# (1, 2) == (1, 2);;
```

```
- : bool = false
```

as in Java

equality written =, on the contrary, is **structural equality**,
that is, recursive equality descending in sub-terms

```
# (1, 2) = (1, 2);;
```

```
- : bool = true
```

it is equivalent to `equals` in Java (when suitably defined)

exceptions

usual notion

an exception may be raised

```
  let division n m =
    if m = 0 then raise Division_by_zero else ...
```

and later caught

```
  try division x y with Division_by_zero -> (0,0)
```

one can introduce new exceptions

```
exception Error
exception Unix_error of string
```

exceptions

usual notion

an exception may be raised

```
let division n m =
  if m = 0 then raise Division_by_zero else ...
```

and later caught

```
try division x y with Division_by_zero -> (0,0)
```

one can introduce new exceptions

```
exception Error
exception Unix_error of string
```

exceptions

usual notion

an exception may be raised

```
  let division n m =
    if m = 0 then raise Division_by_zero else ...
```

and later caught

```
  try division x y with Division_by_zero -> (0,0)
```

one can introduce new exceptions

```
exception Error
exception Unix_error of string
```

idiom

in OCaml, exceptions are used in the library to signal exceptional behavior

example: `Not_found` to signal a missing value

```
try
    let v = Hashtbl.find table key in
    ...
with Not_found ->
    ...
```

(where Java typically returns `null`)

modules and functors

when programs get big we need to

- split code into units (modularity)
- hide data representation (encapsulation)
- avoid duplicating code

in OCaml, this is provided by modules

files and modules

each file is a module

if arith.ml contains

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

we compile it with

```
% ocamlopt -c arith.ml
```

we use it within another module main.ml:

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

files and modules

each file is a module

if `arith.ml` contains

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

we compile it with

```
% ocamlopt -c arith.ml
```

we use it within another module `main.ml`:

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

files and modules

each file is a module

if arith.ml contains

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

we compile it with

```
% ocamlopt -c arith.ml
```

we use it within another module main.ml:

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

files and modules

each file is a module

if arith.ml contains

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

we compile it with

```
% ocamlopt -c arith.ml
```

we use it within another module main.ml:

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

files and modules

each file is a module

if arith.ml contains

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

we compile it with

```
% ocamlopt -c arith.ml
```

we use it within another module main.ml:

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

encapsulation

we can limit what is exported with an interface

in a file arith.mli

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

we can limit what is exported with an interface
in a file arith.mli

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

we can limit what is exported with an interface

in a file arith.mli

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

we can limit what is exported with an interface
in a file arith.mli

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

an interface may also hide the definition of a type

in set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

but in set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

type t is an abstract type

encapsulation

an interface may also hide the definition of a type

in set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

but in set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

type t is an abstract type

encapsulation

an interface may also hide the definition of a type

in set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

but in set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

type t is an abstract type

encapsulation

an interface may also hide the definition of a type

in `set.ml`

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

but in `set.mli`

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

type `t` is an abstract type

separate compilation

the compilation of a file only depends on the interfaces of the other files

$\Rightarrow$ fewer recompilation when a code changes but its interface does not

separate compilation

the compilation of a file only depends on the interfaces of the other files
$\Rightarrow$ fewer recompilation when a code changes but its interface does not

module system

not limited to files

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

module system

not limited to files

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

module system

not limited to files

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

module system

similar for interfaces

```
module type S = sig
  val f : int -> int
end
```

interface constraint

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

module system

similar for interfaces

```
module type S = sig
  val f : int -> int
end
```

interface constraint

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

module system

similar for interfaces

```
module type S = sig
  val f : int -> int
end
```

interface constraint

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

module system

similar for interfaces

```
module type S = sig
  val f : int -> int
end
```

interface constraint

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

recap

- code split into units called modules
- encapsulation of types and values, abstract types
- separate compilation
- organizes the name space

functors

functor = module parameterized with other modules

example: hash table

one has to parameterize wrt hash function and equality function

functors

functor = module parameterized with other modules

example: hash table
one has to parameterize wrt hash function and equality function

the solution: a functor

```
module type HashedType = sig
  type elt
  val hash: elt -> int
  val eq  : elt -> elt -> bool
end
```

```
module HashTable(X: HashedType) = struct ... end
```

functor definition

```
module HashTable(X: HashedType) = struct
  type t = X.elt list array
  let create n = Array.make n []
  let add t x =
    let i = (X.hash x) mod (Array.length t) in
    t.(i) <- x :: t.(i)
  let mem t x =
    let i = (X.hash x) mod (Array.length t) in
    List.exists (X.eq x) t.(i)
end
```

inside, `X` is used as any regular module

functor type

```
module HashTable(X: HashedType) : sig
  type t
  val create : int -> t
  val add : t -> X.elt -> unit
  val mem : t -> X.elt -> bool
end
```

functor use

```
module Int = struct
  type elt = int
  let hash x = abs x
  let eq x y = x=y
end
```

```
module Hint = HashTable(Int)
```

```
# let t = Hint.create 17;;
```

```
val t : Hint.t = <abstr>
```

```
# Hint.add t 13;;
```

```
- : unit = ()
```

```
# Hint.add t 173;;
```

```
- : unit = ()
```

functor use

```
module Int = struct
  type elt = int
  let hash x = abs x
  let eq x y = x=y
end
```

```
module Hint = HashTable(Int)
```

```
# let t = Hint.create 17;;
```

```
val t : Hint.t = <abstr>
```

```
# Hint.add t 13;;
```

```
- : unit = ()
```

```
# Hint.add t 173;;
```

```
- : unit = ()
```

functor use

```
module Int = struct
  type elt = int
  let hash x = abs x
  let eq x y = x=y
end
```

```
module Hint = HashTable(Int)
```

```
# let t = Hint.create 17;;
```

```
val t : Hint.t = <abstr>
```

```
# Hint.add t 13;;
```

```
- : unit = ()
```

```
# Hint.add t 173;;
```

```
- : unit = ()
```

parallel

Java	OCaml
`interface HashedType<T> {`	`module type HashedType = sig`
	`   type elt`
`   int hash();`	`   val hash:  elt -> int`
`   boolean eq(T x);`	`   val eq:  elt -> elt -> bool`
`}`	`end`
`class HashTable`	`module HashTable(E: HashedType) =`
`   <E extends HashedType<E>> {`	`struct`
`      ...`	`   ...`

applications of functors

1. data structures parameterized with other data structures
   - `Hashtbl.Make` : hash tables
   - `Set.Make` : finite sets implemented with balanced trees
   - `Map.Make` : finite maps implemented with balanced trees

2. algorithms parameterized with data structures

example: Dijkstra's algorithm

```
module Dijkstra
  (G: sig
        type graph
        type vertex
        val succ: graph -> vertex -> (vertex * float) list
      end) :
  sig
  val shortest_path:
    G.graph -> G.vertex -> G.vertex -> G.vertex list * float
  end
```

persistence

immutable data structures

in OCaml, most data structures are **immutable**
(exceptions are arrays and records with `mutable` fields)

said otherwise:

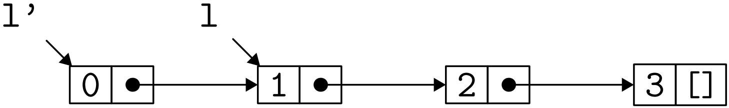
- a value is not modified by an operation,
- but a **new** value is returned

terminology: this is called **applicative programming** or **functional programming**

example of immutable structure: lists

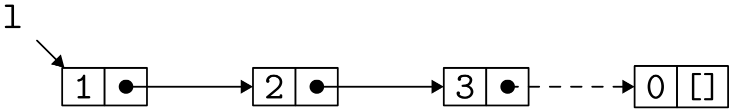
```
let l = [1; 2; 3]
```

```
let l' = 0 :: l
```



no copy, but sharing

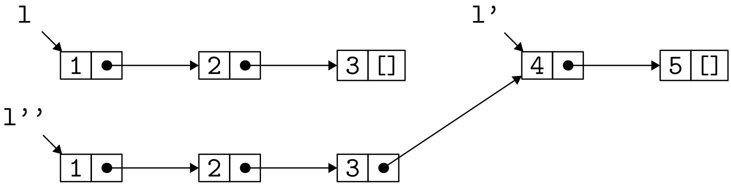
adding an element at the end of the list is not simple:



concatenating two lists

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l'
```

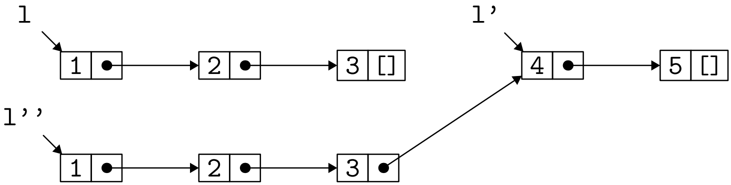


blocs of `l` are copied, blocs of `l'` are shared

concatenating two lists

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```

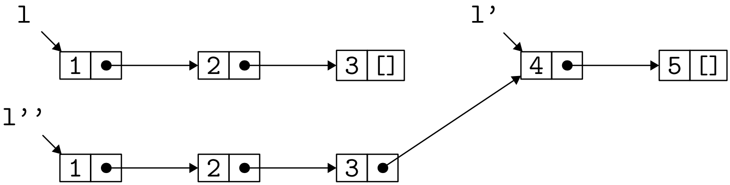


blocs of `l` are copied, blocs of `l'` are shared

concatenating two lists

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs of `l` are copied, blocs of `l'` are shared

mutable linked lists

note: one can implement traditional linked lists,
for instance with

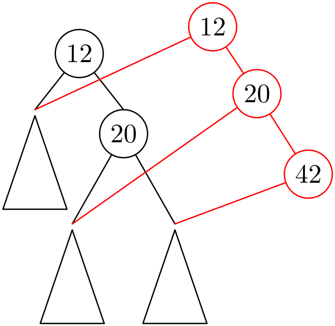
```
type 'a mlist = Empty | Element of 'a element
and 'a element = { value: 'a; mutable next: 'a mlist }
```

but then be careful with **sharing** (*aliasing*)

another example: trees

```
type tree = Empty | Node of int * tree * tree

val add : int -> tree -> tree
```



again, few copies and mostly sharing

benefits of persistence

1. correctness of programs
   - code is simpler
   - mathematical reasoning is possible

2. easy to perform backtracking
   - search algorithms
   - symbolic manipulation and scopes
   - error recovery

benefits of persistence

1. correctness of programs
    - code is simpler
    - mathematical reasoning is possible
2. easy to perform backtracking
    - search algorithms
    - symbolic manipulation and scopes
    - error recovery

persistence and backtracking (1)

search for a path in a maze

```
type state
val is_exit : state -> bool
type move
val moves : state -> move list
val move : state -> move -> state
```

```
let rec search e =
  is_exit e || iter e (moves e)
and iter e = function
  | []     -> false
  | d :: r -> search (move d e) || iter e r
```

persistence and backtracking (1)

search for a path in a maze

```
type state
val is_exit : state -> bool
type move
val moves : state -> move list
val move : state -> move -> state
```

```
let rec search e =
  is_exit e || iter e (moves e)
and iter e = function
  | []      -> false
  | d :: r -> search (move d e) || iter e r
```

persistence and backtracking (1)

search for a path in a maze

```
type state
val is_exit : state -> bool
type move
val moves : state -> move list
val move : state -> move -> state
```

```
let rec search e =
  is_exit e || iter e (moves e)
and iter e = function
  | []     -> false
  | d :: r -> search (move d e) || iter e r
```

without persistence

with a mutable, global state

```
let rec search () =
  is_exit () || iter (moves ())
and iter = function
  | []      -> false
  | d :: r -> (move d; search ()) || (undo d; iter r)
```

i.e. one has to **undo** the side effect
(here with a function `undo`, inverse of `move`)

persistence and backtracking (2)

simple Java fragments, represented with

```
type stmt =
  | Return of string
  | Var    of string * int
  | If     of string * string * stmt list * stmt list
```

example:

```
int x = 1;
int z = 2;
if (x == z) {
  int y = 2;
  if (y == z) return y; else return z;
} else
  return x;
```

persistence and backtracking (2)

let us check that any variable which is used was previously declared (within a list of statements)

```
val check_stmt : string list -> stmt -> bool
val check_prog : string list -> stmt list -> bool
```

persistence and backtracking (2)

```
let rec check_instr vars = function
  | Return x ->
      List.mem x vars
  | If (x, y, p1, p2) ->
      List.mem x vars && List.mem y vars &&
      check_prog vars p1 && check_prog vars p2
  | Var _ ->
      true
```

```
and check_prog vars = function
  | [] ->
      true
  | Var (x, _) :: p ->
      check_prog (x :: vars) p
  | i :: p ->
      check_instr vars i && check_prog vars p
```

persistence and backtracking (2)

```
let rec check_instr vars = function
  | Return x ->
      List.mem x vars
  | If (x, y, p1, p2) ->
      List.mem x vars && List.mem y vars &&
      check_prog vars p1 && check_prog vars p2
  | Var _ ->
      true
```

```
and check_prog vars = function
  | [] ->
      true
  | Var (x, _) :: p ->
      check_prog (x :: vars) p
  | i :: p ->
      check_instr vars i && check_prog vars p
```

persistence and backtracking (3)

a program handles a database

non atomic updates, requiring lot of computation

with a mutable state

```
try
  ... performs update on the database ...
with e ->
  ... rollback database to a consistent state ...
  ... handle the error ...
```

persistence and backtracking (3)

a program handles a database

non atomic updates, requiring lot of computation

with a mutable state

```
try
  ... performs update on the database ...
with e ->
  ... rollback database to a consistent state ...
  ... handle the error ...
```

persistence and backtracking (3)

a program handles a database

non atomic updates, requiring lot of computation

with a mutable state

```
try
  ... performs update on the database ...
with e ->
  ... rollback database to a consistent state ...
  ... handle the error ...
```

persistence and backtracking (3)

with a persistent data structure

```
let bd = ref (... initial database ...)
...
try
  bd := (... compute the update of !bd ...)
with e ->
  ... handle the error ...
```

interface and persistence

the persistent nature of a type is not obvious

the signature provides implicit information

mutable data structure

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

persistent data structure

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

interface and persistence

the persistent nature of a type is not obvious
the signature provides implicit information

mutable data structure

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

persistent data structure

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

interface and persistence

the persistent nature of a type is not obvious
the signature provides implicit information
mutable data structure

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

persistent data structure

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

interface and persistence

the persistent nature of a type is not obvious

the signature provides implicit information

mutable data structure

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

persistent data structure

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

persistence and side effects

persistence does not mean absence of side effects

$$\textit{persistent} = \textit{observationally immutable}$$

only one way

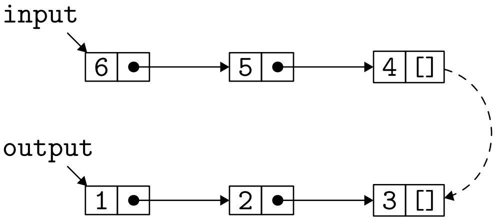
$$\textit{immutable} \Rightarrow \textit{persistent}$$

the reciprocal is wrong

example: persistent queues

```
type 'a t
val create : unit -> 'a t
val push : 'a -> 'a t -> 'a t
exception Empty
val pop : 'a t -> 'a * 'a t
```

idea: a queue is a **pair of lists**,
one for insertion, and one for extraction



stands for the queue $\rightarrow 6, 5, 4, 3, 2, 1 \rightarrow$

example: persistent queues

```
type 'a t = 'a list * 'a list

let create () = [], []

let push x (e,s) = (x :: e, s)

exception Empty

let pop = function
  | e, x :: s -> x, (e,s)
  | e, [] -> match List.rev e with
    | x :: s -> x, ([], s)
    | [] -> raise Empty
```

example: persistent queues

when accessing several times the same queue whose second list is empty, we reverse several times the same list

let's add a reference to register the list reversal the first time it is performed

```
type 'a t = ('a list * 'a list) ref
```

the side effect is done "under the hood", in a way not observable from the user, the contents of the queue staying the same

example: persistent queues

```
let create () = ref ([], [])

let push x q = let e,s = !q in ref (x :: e, s)

exception Empty
```

```
let pop q = match !q with
  | e, x :: s -> x, ref (e,s)
  | e, [] -> match List.rev e with
      | x :: s as r -> q := [], r; x, ref ([], s)
      | [] -> raise Empty
```

example: persistent queues

```
let create () = ref ([], [])

let push x q = let e,s = !q in ref (x :: e, s)

exception Empty
```

```
let pop q = match !q with
  | e, x :: s -> x, ref (e,s)
  | e, [] -> match List.rev e with
      | x :: s as r -> q := [], r; x, ref ([], s)
      | [] -> raise Empty
```

recap

- persistent structure = no observable modification
  - in OCaml: `List`, `Set`, `Map`
- can be very efficient (lot of sharing, hidden side effects, no copies)
- idea independent of OCaml