《编译原理》

第2讲: OCaml简介

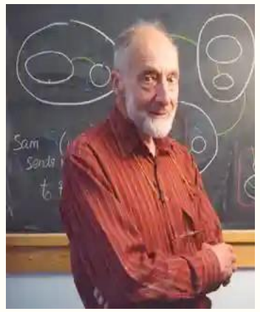
何冬杰
重庆大学

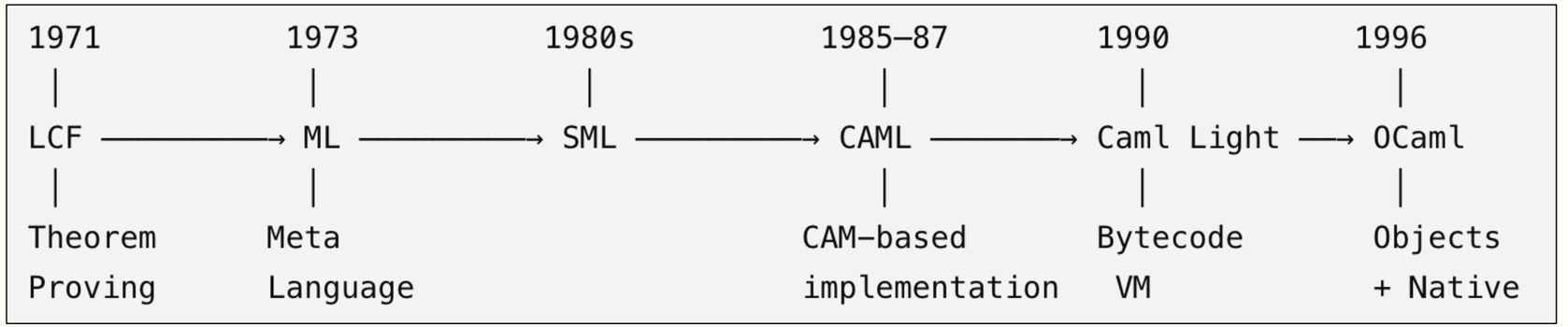
OCaml

```
# brew install opam
# opam init
# opam switch create ocaml-latest
```

https://ocaml.org/docs/installing-ocaml

- OCaml: Objective Categorical Abstract Machine Language
- A brief history


Robin Milner




Xavier Leroy

- 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 by **Xavier Leroy** and others at Inria Rocquencourt, France
- Some applications:
  - symbolic computation and languages (IBM, Intel,Dassault Syst`emes),
  - 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!
```

Variables and References

- let x = e introduces a global **variable**
- Differences with respect to usual notion of variable:
  - necessarily initialized
  - type not declared but inferred
  - cannot be assigned

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

- 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
```

“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)
```

- Local variables:
  - In C or Java, local variables appear in a block:

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

  - In OCaml, a local variable is introduced with **let in** as for a global variable:
    - Necessarily initialized, type inferred, immutable,
    - But scope limited to the expression following **in**

Programs

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

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

- Examples

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

unit type

- Expressions with no meaningful value have type unit
  - Expressions include assignment, loop, ...
  - This type has a single value, written ()
- Example:
  - 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)
```

Interactive Loop

- Interactive version of the compiler: ocaml, **utop** (preferred)

```
[➜  ~ ocaml
OCaml version 5.4.1
Enter #help;; for help.

# let x = 1 in x + 2;;
- : int = 3
# let y = 1 + 2;;
val y : int = 3
# y * y;;
- : int = 9
# 
```

```
        Welcome to utop version 2.17.0 (using OCaml version 5.4.1)!

Type #utop_help for help about using utop.

─( 18:06:13 )─< command 0 >──────────────────────────────{ counter: 0 }─
utop # let x = 1 in x + 2;;
- : int = 3
─( 18:06:13 )─< command 1 >──────────────────────────────{ counter: 0 }─
utop # let y = 1 + 2;;
val y : int = 3
─( 18:06:25 )─< command 2 >──────────────────────────────{ counter: 0 }─
utop # y*y;;
- : int = 9
─( 18:06:32 )─< command 3 >──────────────────────────────{ counter: 0 }─
utop # 
Arg  Array  ArrayLabels  Assert_failure  Atomic  Bigarray  Bool  Buffer  Bytes  Byte
```

Function syntax

- Example:

```
# let f x = x * x;;
val f : int -> int = <fun>
# f 4;;
- : int = 16
```

  - Body = expression (no return)
  - Type is inferred (types of argument x and result)

- Java vs OCaml

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
```

Function with/without arguments

- Function without arguments:
  - Takes an argument of type unit

```
# 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
```

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;;
```

```
- : bool -> true
```

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;;
```

identical ⟺

```
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>

# g 4;;
```

```
# let g = f 3;;
val g : int -> int = <fun>

- : int = 25
```

- 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 = 0.45548650838731830 1
```

```
# 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: (element_type -> unit) -> set -> unit
```

  - Example:

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

Difference w.r.t. 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**
  - closure = function + environment

- Note: there are closures in Java (≥ 8) too

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

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 = ()
```

- 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>
```

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|]
```

Memory Allocation

- Memory allocation handled by a garbage collector (GC)
- GC benefits:
  - unused memory is reclaimed automatically
  - efficient allocation
- Forget about “dynamic allocation is expensive”
  - but keep worrying about complexity!

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

Recursive functions

- In OCaml, it is idiomatic to use recursive functions
  - a function call is cheap, and tail calls are optimized
  - Recursive code: clearer, simpler to justify
- Example: Insertion Sort

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

```
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
```

```
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.
```

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
```

- Revisit 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

Class in Java vs Record in OCaml

Java

```
class T {
        final int v; boolean b;
        T(int v, boolean b) {
                                this.v = v; this.b = b;
        }
}

T r = new T(42, true);

r.b = false;

r.v
```

OCaml

```
  type t = {
        v:  int;
        mutable b:  bool;
}


Let r = { v = 42; b = true }

r.b <- false

r.v
```

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
```

Lists

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

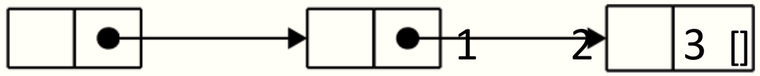
```
# let l = 1 :: 2 :: 3 :: [];;
```

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

  - Shorter Syntax

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

- Presentation in memory
  - Ocaml lists = identical to lists in C or Java
  - The list [1; 2; 3] is represented as



Algebraic Data Types (ADT)

- 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 = 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 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>
```

- Patterns can be nested, omitted or grouped

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

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

- pattern matching is not limited to algebraic data types
  - one may write let pattern = expression when there is a single pattern (as in let (a,b,c,d) = v for instance)

Inheritance in Java vs ADT in OCaml

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

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


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

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

- A **value**, which fits on 64 bits, is
  - either a *primitive value* (integer, floating point, Boolean, [], etc.)
  - or a *pointer* (to an array, a constructor such as And, etc.)
- In OCaml, there is no null value
  - Any value is necessarily initialized
  - no such thing as NullPointerException in Java
  - neither segmentation fault as in C/C++
  - 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
- In OCaml, passing mode is by value
  - In particular, no value is ever copied
  - It is exactly as in Java

Comparison

- equality written == is physical equality,
  - that is, equality of pointers or primitive values as in Java

```
#  (1, 2) ==  (1, 2);;
-  : bool = false
```

- 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

- 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 ParseError of string
```

- In OCaml, exceptions are used in the library to signal exceptional behavior
  - example: Not found to signal a missing value, where Java typically returns null

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

Modules

- In software engineering, 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
  - In OCaml, each **file** is a **module**
  - if arith.ml contains

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

  - we can use it within another module main.ml:

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

Encapsulation

- We can limit what is exported with an interface
  - in file arith.mli

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

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

  - Arith.pi cannot be found as it is not exported in the interface
- An interface may also hide the definition of a type

set.ml

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

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

- Compilation of a file only depends on the interfaces of the other files
  - fewer recompilation when a code changes but its interface does not

Module System

- We can also use keyword **module** to define a module

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

```
module A = struct
    module B = struct
        let f x = 6 * x
    end
    let f x = B.f (x + 1)
end
```

- We also define **interfaces** for modules

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

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

Inferface constraint

M.a is hidden

- Keyword **open** for namespace import: “open Arith” allows use pi directly

inside, X is used as any regular module

Functors

- functor = module parameterized with other modules
- Example: Hash Table

```
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
```

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

```
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 type

Functor

❑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 = ()
```

❑Applications of functors

- data structures parameterized with other data structures
  - Hashtbl.Make : hash tables
  - Set.Make : finite sets via balanced trees
  - Map.Make : finite maps via balanced trees
- algorithms parameterized with data structures

```
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
```

HashTable implementation in Java/OCaml

Java

```
interface HashedType<T> {

      int hash();
      boolean eq(T x);
}

class HashTable
      <E extends HashedType<E>> {  …
```

OCaml

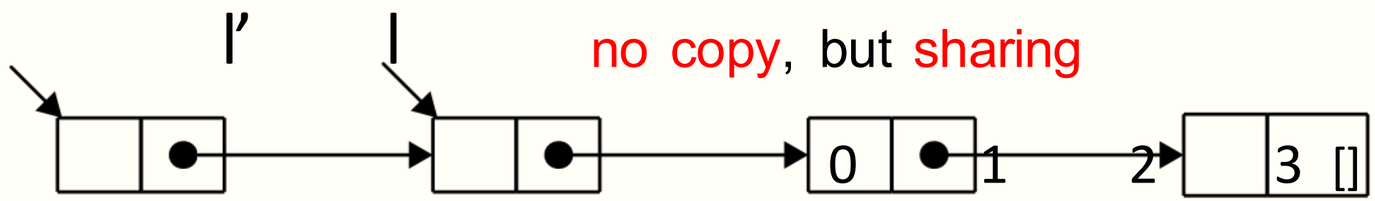
```
module type HashedType =  sig
      type elt
      val hash:  elt -> int
      val eq:   elt -> elt -> bool
end

module HashTable(E: HashedType) =
struct
      …
```

Persistence

- Immutable data structures
  - a value is not modified by an operation,
  - but a new value is returned
  - this is called applicative programming or functional programming
- In OCaml, most data structures are **immutable**
  - exceptions are arrays and records with mutable fields
- Example of immutable structure: lists

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

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

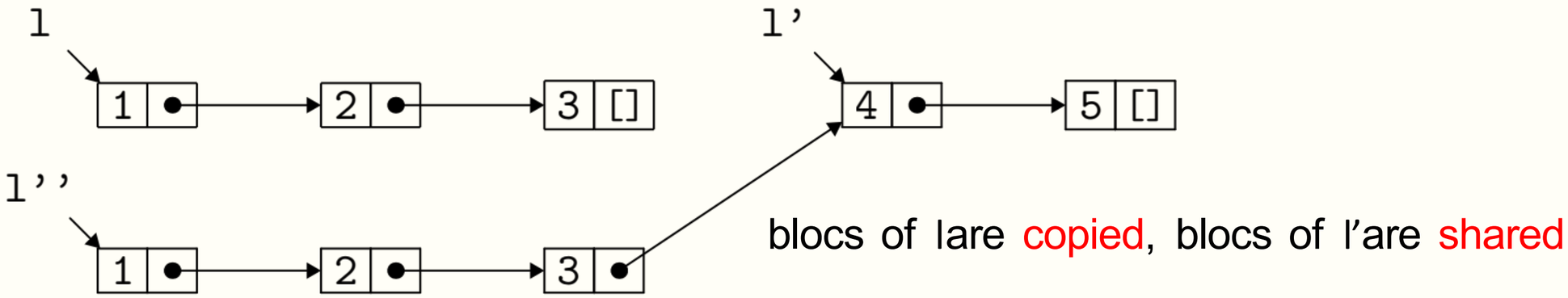


Example of immutable structure: lists

- Concatenating two lists:

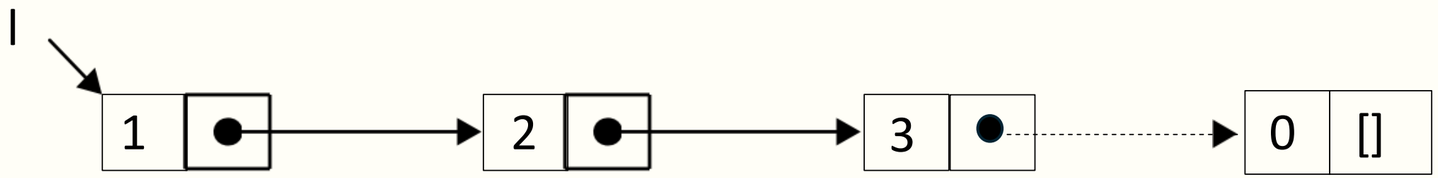
```
let rec append l1 l2 = match l1 with
    |  [] -> l2
    |  x  :: l -> x  :: append l l2
```

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



Example of immutable structure: lists

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



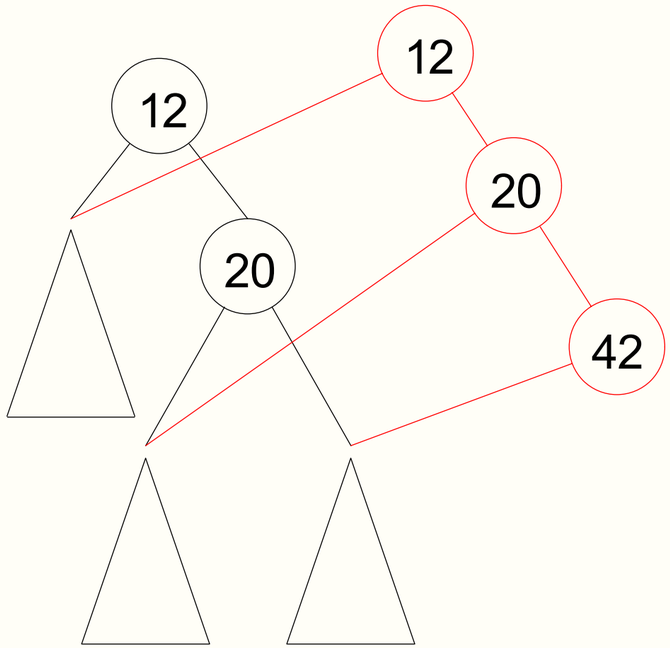
- Mutable linked lists
  - One can implement traditional linked lists, for instance with

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

  - but then be careful with sharing (aliasing)

Example of immutable structure: trees

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

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



Insert an element

Again, few copies and mostly sharing!

Persistence and Backtracking

❑ Example A:

- 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 e d)
                || 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: Example B

- Simple Java fragments, represented with

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

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

Example code

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

```
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: Example C

- a program handles a database
  - non atomic updates, requiring lot of computation
- with a mutable state

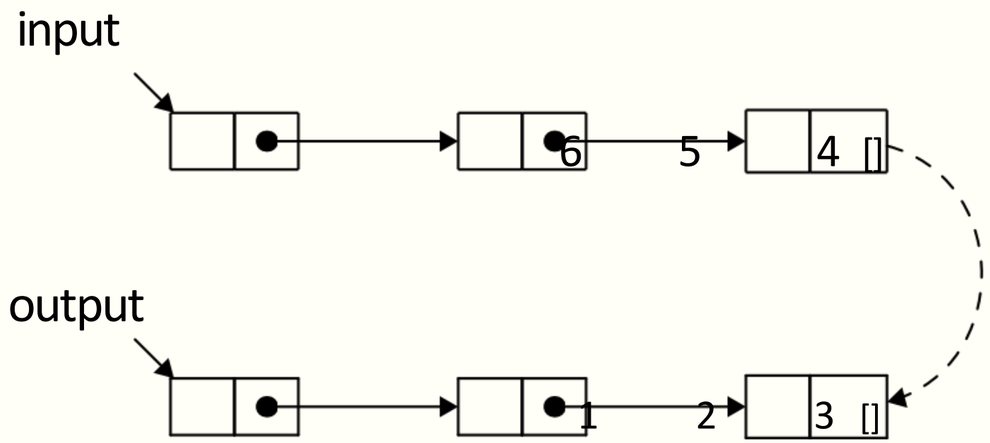
```
try
    ... performs update on the database ...
with e ->
    ... rollback database to a consistent state ...
    ... handle the error ...
```

- with a persistent data structure

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

Example of immutable structure: persistent queues

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



stands for the queue → 6, 5, 4, 3, 2, 1 →

- Define **persistent queue**

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

```
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 of immutable structure: 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

- persistent = observationally immutable
  - Persistence does not mean absence of side effects
  - immutable → persistent
  - the reciprocal is wrong

```
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
```

Benefits of Persistence

- 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
- Benefits of Persistence
  - Correctness of programs
    - code is simpler
    - mathematical reasoning is possible
  - Easy to perform backtracking
    - search algorithms
    - symbolic manipulation and scopes error recovery

Ocaml Tools

- opam — package manager
- utop — a more fully-featured interactive top-level
  - ~~ocaml — the top-level interactive loop~~
- ocamlc — the bytecode compiler
- ocamlopt — the native code compiler
- ocamllex — the lexer generator
- menhir — a more modern parser generator
  - ~~ocamlyacc — the parser generator~~
- dune — build system
  - ~~ocamlbuild — a compilation manager~~

References

- A Short Introduction to OCaml, Jean-Christophe Filliˆatre, https://www.enseignement.polytechnique.fr/profs/informatique/Jean-Christophe.Filliatre/14-15/INF549/slides.pdf
- OCaml Documentation, https://ocaml.org/docs

关于Ocaml语言的任何问题，多问问AI大模型！也欢迎在群里讨论交流

作业

❑1. List Processing

给定成绩列表如：let scores = [45; 82; 60; 92; 39], 使用List.map, List.filter, List.fold_left等完成如下函数：

```
val passed : int list -> int list
val add_bonus : int -> int list -> int list
val average : int list -> float
```

❑2. Expression Evaluator

定义如下类型：

```
type expr = Int of int | Var of string | Add of expr * expr | Let of string * expr * expr
type env = (string * int) list
```

实现如下函数：

```
val lookup : string -> env -> int option
val eval : env -> expr -> int option
```

满足样例：

```
eval [] (Add (Int 3, Int 4))  (* => Some 7 *)
eval [] (Let (“x”, Int 10, Add (Var “x”, Int 5)))  (* => Some 15 *)
eval [] ( Let (“x”, Int 10, Let (“x”, Int 20, Add (Var “x”, Int 1))) )  (* => Some 21 *)
```