A Quick Introduction to OCaml

Lorenzo Ceragioli

September 26, 2024

IMT Lucca

Functional Languages

- Programs are constructed by applying and composing functions
- Functional Languages are declarative: function definitions are expressions (mapping values to values)
- Heavily based on expressions
- Roughly abstracting away from the memory
- Functions are as every other value: first-class citizens
- In contrast to imperative languages where
  - Expressions are only a small part of the constructs
  - The programmer defines and uses commands for updating the memory

Pure Functions

- In a *pure* functional language, functions are mathematical functions (i.e. same input implies same output)
- Functional languages are often enriched with *imperative features*, i.e. not all functions are mathematical functions.
- For example for generating random numbers
- We will mostly deal with the pure fragment of OCaml, please avoid imperative features unless they are really needed!

How to Program in a Pure Functional Languages

- Use recursion instead of loops
- Recall that functions are first class citizens: you can have them as input and outputs of your functions
- Lots of library functions are like this: use them!
- No assignments implies no value can be updated: you usually create a new value each time
- The compiler is smart enough to avoid copying the data structure and just updates it if it is legal

OCaml Basics

REPL - utop

OCaml Read Evaluate Print Loop

```
user@machine:~$ utop
# 5 + 4;;        (* expression *)
- : int = 9      (* its type and value *)

# let s = "hello world";; (*  expression definition *)
val s : string = "hello world"
(*  its name, type and value *)

# print_endline s;;    (*  command  *)
hello world            (*  executing it *)
- : unit = ()          (*  returned type and value *)
```

Hello World and Comments

The program `helloworld.ml`:

```
let () = print_endline "Hello, World!"
(* this is a multiline
comment *)
```

Compiling it:

```
$ ocamlc -o main main.ml
```

Executing it:

```
$ ./main
Hello, World!
```

... but it is simpler to use *dune*!

Some basic types ...

Type	Name	Values
Integer numbers	`int`	1, 2, 3, . . .
Boolean	`bool`	`true`, `false`
String	`string`	`"a"`, `"b"`, `"aa"`, . . .
Unit type	`unit`	`()`

Note: There is no implicit conversion, e.g. from `int` to `string`

And operators

Integer arithmetic	`+, -, *, /, mod`	
Boolean operators	`not, &&, ||`	
Structural Equivalence	`=, <>`	
Physical Equivalence	`==, !=`	← avoid this
Comparison	`<, >, <=, >=`	
String append	`^`	

If-then-else

$$\texttt{if}\ \mathit{expr}_1\ \texttt{then}\ \mathit{expr}_2\ \texttt{else}\ \mathit{expr}_3$$

Note:

- Choose between expressions, not commands or blocks
- The `else` case is mandatory!
- expr_1 must evaluate to a boolean (strongly inferred types!)
- expr_2 and expr_3 must have the same (inferred) type

```
# if "a" = "a" then 42 else 17;;
- : int = 42
```

let and let ... in

Defining names **globally** : `let` *name* `=` *expr*

Defining names **locally** : `let` *name* `=` $expr_1$ `in` $expr_2$

```
# let a =
    let b = "ciao" in
      if b <> "ciao" then 42 else 17;;
val a : int = 17

# print_int a;;
17
- : unit = ()

# print_string b;;
Error: Unbound value x
```

let is not an assignment

Note: you can redefine the value of a name based on the previous name, but it is not an assignment as in imperative languages

```
# let a = 4;;

# let get_a x = a;;

# let a = 4 + a;;
val a : int = 8

# get_a 3;;
- : int = 4
```

The name a is really just a name for a value!

Subsequent definitions update the names, not the values!

Simple Functions (finally having some fun!)

Functions are values in OCaml, i.e. fully evaluated expressions

```
# fun x -> x + 1;;
- : int -> int = <fun>  (* inferred type (no code) *)

# (fun x -> x + 1) 5;;  (* fun application *)
-: int = 6

# let inc = fun x -> x + 1;;  (* naming a function *)
val inc : int -> int = <fun>

# let inc x = x + 1;;  (* same as before *)
val inc : int -> int = <fun>
```

Mind the definition order!

Functions are defined in order

```
# let inc x = (dec x) + 2
let dec x = x - 1;;
Error: Unbound value dec

# let dec x = x - 1
let inc x = (dec x) + 2;;
val dec : int -> int = <fun>
val inc : int -> int = <fun>
```

Polymorphic Functions

In OCaml types are inferred, what if multiple types are possible?

```
# let id x = x
val id : 'a -> 'a = <fun>
```

- The identity function works with every possible type
- 'a is a type variable, it stands for any type
- Note that the return type is also any type, but it coincides with the input type
- We will see polymorphic types that are not "any type"

Recursive Functions

Really just use the `rec` keyword and you can use the function name inside the function code

```
# let rec bang x =            (* naming the function *)
    if x = 0 then 1           (* base case *)
    else x * (bang (x-1));;     (* recursion *)
val bang : int -> int = <fun>
```

Mind that recursive functions may diverge, e.g. `bang -1`

Mutually Recursive Functions

Recall: functions must be defined before they are used in the body of other functions

If you want `f` to be defined in terms of `g` and vice-versa, use `rec` and the **`and`** keywords

```
# let rec f x =              (* defines f *)
    if x < 1 then true
    else g (x-1)             (* uses g *)
  and g x =                  (* defines g *)
    if x < 1 then false
    else f (x-1);;           (* uses f *)
val f : int -> bool = <fun>
val g : int -> bool = <fun>
```

What do `f` and `g` compute?

Simple modules: .ml and .mli files

Idea: Split the code into modules and to hide the implementation.

The interface define the visible values promised by the module

`modulename.mli` ← the interface
`val inc : int -> int`

The implementation must satisfy the promises

`modulename.ml` ← the implementation

```
let dec x = x - 1          (* not visible *)
let inc x = (dec x) + 2 (* visible *)
```

Accessing the module's function from outside

```
let two = Modulename.inc 1
```

Compiling Modules (better to use dune)

modulename.mli ← the interface

```
val inc : int -> int
```

modulename.ml ← the implementation

```
let dec x = x - 1          (* not visible *)
let inc x = (dec x) + 2 (* visible *)
```

main.ml ← the program

```
let () = print_int Modulename.inc 1
```

```
$ ocamlc -c modulename.ml  <-- (creates modulename.cmi)
$ ocamlc -c modulename.ml  <-- (creates modulename.cmo)
$ ocamlc -c main.ml  <-- (creates main.cmo)
$ ocamlc -o ex modulename.cmo main.cmo <-- (creates ex)
$ ./ex
2
```

Types

Pairs

Ordered pairs of elements of possibly different types

```
# let a = (4,"ciao");;
val a : int * string = (4, "ciao")

# fst a;;
- : int = 4

# snd a;;
- : string = "ciao"

# fst;;
- : 'a * 'b -> 'a = <fun>

# let sum (x,y) = x + y;;
val sum : int * int -> int = <fun>
```

Tuples

Ordered collections of elements of possibly different types (pairs are a special case)

```
# let b = (1,5,(6,true));;
val b : int * int * (int * bool) = (1, 5, (6, true))

# fst b;;
Error:This expression has type int * int * (int * bool)
        but an expression was expected of type 'a * 'b

# fst;;    (* fst and snd only defined on pairs *)
- : 'a * 'b -> 'a = <fun>
```

Note: the type depends on the length of the tuple!

Lists

```
# [];;             (* Empty list ... *)
- : 'a list = []   (* ... is polymorphic! *)

# [1];;
- : int list = [1]

# [1;2;3;4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

# ["a";4];;
 Error: This expression has type int but an expression
   was expected of type string
```

Note: all elements must be of the same type

Note: the type do NOT depend on the length

Lists/Tuple confusion

```
# [(4,5);(6,7);(8,9)];;
- : (int * int) list = [(4, 5); (6, 7); (8, 9)]

# [(4,5)];;
- : (int * int) list = [(4, 5)]

# [4,5];;
- : (int * int) list = [(4, 5)]

# 4,5;;  (* because parenthesis are optional! *)
- : int * int = (4, 5)

# [4,5,6,7];;
- : (int * int * int * int) list = [(4, 5, 6, 7)]
```

Note: use **semicolon** for lists!

Operations on Lists

Operation	Example	Note
Cons	`4::[1;2;3] = [4;1;2;3]`	fast
Concat	`[4;5;6]@[1;2;3] = [4;5;6;1;2;3]`	$O(n)$
Head	`List.hd [4;5;6] = 4`	fast
Tail	`List.tl [4;5;6] = [5;6]`	fast
Reverse	`List.rev [4;5;6] = [6;5;4]`	$O(n)$

Note: `List` is the module of lists, several useful functions other than `hd`, `tl`, and `rev`.

User Defined Types

- Users can define their own types with the `type` construct
- Types can be recursive (no need for `rec` keyword)
- Polymorphic types defined through *type parameters*
- Some examples follow

Records

Idea: like tuples but with names

```
# type point = { x : int; y : int; z : int };;
type point = { x : int; y : int; z : int; }

# let origin = { x = 0; y = 0; z = 0 };;
val origin : point = {x = 0; y = 0; z = 0}

(* a new point with some changed values *)
# let p = { origin with x = 10; y = 5 };;
val p : point = {x = 10; y = 5; z = 0}

# p.x;;
- : int = 10
```

Note: must be explicitly defined as types!

Variants

Elements can ne of either of the given (named) shapes.

```
type qual_temp = Hot | Cold | Fine

type temp =
  Precise of int
  | Approx of qual_temp
```

- `qual_temp` is an *enumeration* of atomic values
- `temp` is either a `qual_temp` (tagged as Approx) or a precise int for the degrees (tagged as Precise)
- Tags must be globally unique and start with a capital letter

Recursive Variants

Arithmetic expressions

```
type a_exp =
  Aval of int
  | Plus of a_exp * a_exp
  | Minus of a_exp * a_exp
  | Times of a_exp * a_exp
```

- You can define `exp` in terms of `exp` itself
- This is useful for defining the abstract syntax trees of programming languages

Mutually Recursive Variants

Arithmetic and boolean expressions

```
type a_exp =
  Aval of int
  | Plus of a_exp * a_exp
  | Minus of a_exp * a_exp
  | Times of a_exp * a_exp

type b_exp =
  Bval of bool
  | And of b_exp * b_exp
  | Or of b_exp * b_exp
  | Not of b_exp
  | Minor of a_exp * a_exp
```

- Boolean expressions defined in terms of arithmetic expressions
- What if we want a boolean to be considered also an integer (e.g. `false = 0` and `true = 1` as in C)?

Mutually Recursive Variants

Arithmetic and boolean expressions

```
type a_exp =
  Aval of int
  | Plus of a_exp * a_exp
  | Minus of a_exp * a_exp
  | Times of a_exp * a_exp
  | Of_bool of bexp
and b_exp =
  Bval of bool
  | And of b_exp * b_exp
  | Or of b_exp * b_exp
  | Not of b_exp
  | Minor of a_exp * a_exp
```

Note: we use `and` as in mutually recursive functions

Polymorphic Variants

Like for lists, we can define polymorphic types through type parameters

```
type 'a btree =
  Leaf of 'a
  | Node of 'a btree * 'a btree

let tree1 =
  Node (Leaf 1,
        Node (Leaf 2, Leaf 3))
(* is of type int tree*)

let tree2 =
  Node (Leaf "a",
        Node (Leaf "b", Leaf "c"))
(* is of type string tree*)
```

Pattern Matching

Pattern matching is a form of branching similar to `switch case`

```
let f x =
  match x with
  | 0
  | 1 -> "one or less"
  | 2
  | 3 -> "two or three"
  | _ -> "four or more"
```

Note: the wildcard _ match everything

Pattern Matching Variants

Pattern matching can

- branch based on the syntactic structure (i.e. how the value is build according to the type definition)
- bind the values of subterms to local names

```
let rec count_leaves x =
  match x with
  | Leaf _ -> 1
      (* I don't care the value inside the leaf *)
  | Node (x, y) ->
      (* x and y are local names for the subterms *)
      (count_leaves x) + (count_leaves y)
```

Note: the compiler will complain if some case is missing

Function Syntax

Pattern matching can be used implicitly with the keyword `function`

```
let rec count_leaves x =
  match x with
  | Leaf _ -> 1
  | Node (x, y) ->
      (count_leaves x) + (count_leaves y)
```

Is the same as

```
let rec count_leaves = function
  | Leaf _ -> 1
  | Node (x, y) ->
      (count_leaves x) + (count_leaves y)
```

Pattern Matching Variants: `as` and `when`

Computing a list of *leaves* containing a value smaller then 10 (the type of the tree is automatically inferred)

```
let rec less_ten_leaves x =
  match x with
  | Leaf n as leaf when n < 10 -> [leaf]
    (* n is the name of the subterm *)
    (* leaf is the name of the whole term *)
    (* therms are matched only when n < 10 *)
  | Leaf _ -> []
    (* all the other kind of leaves, i.e. n >= 10 *)
  | Node (x, y) ->
      (less_ten_leaves x) @ (less_ten_leaves y)
```

Note: sometimes they make your code a lot easier to read!

Pattern Matching with list

The cons operator `::` is not actually an operator, it is a type constructor!

```
# (@);;
- : 'a list -> 'a list -> 'a list = <fun>

# (::);;
Error: The constructor :: expects 2 argument(s),
       but is applied here to 0 argument(s)
```

Hence, we can use it for pattern matching!

Pattern Matching with list

Weird length of a list

```
let rec weird_length ls =
  match ls with
  | [] -> "zero"
  | [_; _] -> "two"
  | _ :: ls' -> "one plus " ^ (weird_length ls')

# weird_length [0;0;0;0;0;0];;
- : string = "one plus one plus one plus one plus two"

# weird_length [0];;
- : string = "one plus zero
```

Exercise 1. Write a pair of functions for evaluating arithmetical and boolean expressions without the `Of_bool` case

Exercise 2. Same but with the `Of_bool` case

Exercise 3. Write a type for general polymorphic trees (i.e. with any number of children)

Exercise 4. Write a function that packs consecutive duplicates of the input list elements into sublists.

For example, the function over [0;0;2;3;3;3;0;2] must return [[0;0];[2];[3;3;3];[0];[2]]

More on Functions

Curried Functions

- Addition usually takes two parameters:

```
let sum (x,y) = x + y
val sum: int * int -> int
```

- In OCaml, one usually writes the curried version (also the kind of functions that you will find in the libraries!)

```
let sum x y = x + y
val sum: int -> int -> int = <fun>
```

- the latter is the same as writing

```
let sum x = fun y -> x + y
val sum: int -> int -> int = <fun>
```

Curried Functions

- Why curried functions? For allowing partial application!

```
let addtwo =
  let sum x y = x + y
  in  sum 2
val sum: int -> int = <fun>

# addtwo 3
- : int = 5
```

- Functions are first class citizens in OCaml, hence they can be the returned value of a function
- A function over functions is called an *higher order function*!

Higher Order Functions

Functions as inputs and/or outputs

```
let twice f x =
  f (f x)
val twice: ('a -> 'a) -> 'a -> 'a = <fun>

# twice not;;
- : bool -> bool = <fun>

let fun_pair f g =
  fun (x, y) -> (f x, g y)
val fun_pair : ('a -> 'b) -> ('c -> 'd)
      -> 'a * 'c -> 'b * 'd = <fun>

# fun_pair not addtwo;;
- : bool * int -> bool * int = <fun>
```

Useful Functions in the List module

`map : ('a -> 'b) -> 'a list -> 'b list`

`map f ls` applies the function `f` to each element of a list to produce another list.

```
# List.map (fun x -> x + 2) [0;2;4;6;8];;
- : int list = [2; 4; 6; 8; 10]
```

Note: if you need the index too, use `List.mapi`

```
# List.mapi (fun i x -> i + x) [0;2;4;6;8];;
(* same as List.mapi (+) [0;2;4;6;8];; *)

- : int list = [0; 3; 6; 9; 12]
```

Useful Functions in the List module

`filter :  ('a -> bool) -> 'a list -> 'a list`

`filter f ls` returns a list containing the elements of `ls` that satisfies the function `f` (preserving the order).

```
# List.filter (fun x -> x mod 2 = 0) [0;3;4;6;9];;
- : int list = [0; 4; 6]
```

Useful Functions in the List module

`fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

`fold_left f initial ls` uses each element of the list to update a value initially instantiated as `initial`; the value is updated using the given function `f`; the result is the one obtained when all the list has been used.

It is the same as `f (...(f (f initial b1) b2)...) bn` when `ls = [b1; b2 ; ...; bn]`.

```
# List.fold_left (fun acc x -> acc + x) 0 [0;1;2;3;4];;
- : int = 10
```

Different Styles of Combining Functions

With explicit parameters

```
let sum_even ls =
  List.fold_left
    (fun acc x -> acc + x)
    0
    (List.filter
      (fun x -> x mod 2 = 0)
      ls)
```

Different Styles of Combining Functions

With intermediate results

```
let sum_even ls =
  let even =
    List.filter
      (fun x -> x mod 2 = 0)
      ls
  in List.fold_left
    (fun acc x -> acc + x)
    0
    even
```

Different Styles of Combining Functions

The (almost) mathematician composition: with the `@@` operator

```
let sum_even ls =
  List.fold_left
    (fun acc x -> acc + x)
    0
  @@
  List.filter
    (fun x -> x mod 2 = 0)
  @@
  ls
```

Note: `@@` stands for function application, because of associativity behaves almost as a function composition $(f \circ g)x = f(g\ x)$, but the input `ls` must be given

Different Styles of Combining Functions

The (almost) computer scientist composition: with the |> operator

```
let sum_even ls =
  ls
  |>
  List.filter
    (fun x -> x mod 2 = 0)
  |>
  List.fold_left
    (fun acc x -> acc + x)
    0
```

Note: |> stands for function pipelining, because of associativity behaves almost as a function composition $(f;g)x = g(f\ x)$, but the input `ls` must be given

Exercises

Exercise 5. Write a function that given two functions f and g return a function over pairs defined as f on the first element and g on the second element.

Exercise 6. Write a function that given a list of integers l1 returns a list l2 of the same length such that each element of l2 in position i is the sum of all the elements in l1 with position less or equal to i.

E.g. The function over [3,6,10,2] returns [3,9,19,21]

Exercise 7. Define a type for Finite Automata and a function for checking if a given string is inside the generated language.

Advanced Features

Ref

OCaml's imperative features (but you should use pure functions)

```
let count = ref 0  (* of type int ref *)

let get_inc () =  (* of type  unit -> int *)
  count := !count + 1;
  !count

# get_inc();;
- : int = 1
# get_inc();;
- : int = 2
# count;;
- : int ref = {contents = 2}
# !count;;
- : int = 2
```

Note: also mutable record fields, arrays, for and while loops.

Nested Modules

You can define a nested module inside a file (modulename.ml), i.e. inside the module Modulename itself.

```
module Username : sig  (* interface of the module *)
  type t
  val of_string : string -> t
  val to_string : t -> string
end = struct    (* implementation of the module *)
  type t = string
  let of_string x = x
  let to_string x = x
end
```

Module Type and Modules

You can split the signature (interface) and its implementation

- Code hiding
- E.g. you can put the signature of `Username` in the interface of the outmost module `modulename.mli`

```
module type ID = sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end

module String_id : ID = struct
  type t = string
  let of_string x = x
  let to_string x = x
end
```

Repetita: Accessing Module Values and Types

We can either qualify the names

```
module type ID = sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end

module String_id : ID = struct
  type t = string
  let of_string x = x
  let to_string x = x
end

# String_id.of_string "ciao"
- : String_id.t = <abstr>
```

Repetita: Accessing Module Values and Types

Or we can open the module (everything is automatically qualified)

```
module type ID = sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end

module String_id : ID = struct
  type t = string
  let of_string x = x
  let to_string x = x
end

open String_id

# of_string "ciao"
- : String_id.t = <abstr>
```

Including a Module

`include`: copies the content of a module into the current one

```
module ID : sig
  type t
  val of_str : string -> t
  val to_str : t -> string
end = struct
  type t = string
  let of_str x = x
  let to_str x = x
end
```

```
module ExtID : sig
  type t
  val of_str : string -> t
  val to_str : t -> string
  val of_int : int -> t
end = struct
  include ID
  let of_int x = x
    |> string_of_int
    |> of_str
end
```

Including a Type Module

`include`: also copies the content of an interface into the current one

```ocaml
module type ID =
sig
  type t
  val of_str : string -> t
  val to_str : t -> string
end

module StrID : ID =
struct
  type t = string
  let of_str x = x
  let to_str x = x
end
```

```ocaml
module type ExtID =
sig
  include ID
  val of_int : int -> t
end

module ExtStrID : ExtID =
struct
  include StrID
  let of_int x = x
    |> string_of_int
    |> of_str
end
```

A module for ordered lists.

```
module type OrdList = sig
  type elem
  type t
  val empty : t
  val to_list : t -> elem list
  val add : elem -> t -> t
end

module IntListOrdList : OrdList = struct
  type elem = int
  type t = elem list
  let empty = []
  let to_list ls = ls
  let rec add l ls = match ls with
    | [] -> [l]
    | l'::ls' -> if (l <= l') then l::l'::ls'
        else l':: (add l ls')
end
```

Functors: map between modules

Build a module from one that satisfies a given interface!

```
module type TotOrd = sig
  type t
  val lesseq : t -> t -> bool
end

module MakeOrdList (Elem : TotOrd) :
  (OrdList with type elem = Elem.t) =  struct
  type elem = Elem.t
  type t = elem list
  let empty = []
  let to_list ls = ls
  let rec add l ls = match ls with
    | [] -> [l]
    | l'::ls' -> if (Elem.lesseq l l' )
      then l::l'::ls' else l':: (add l ls')
end
```

Using Functors

```
module OrdInt : TotOrd = struct
  type t = int
  let lesseq n1 n2 = n1 <= n2
end

module IntOrdList : (OrdList with type elem = int) =
    MakeOrdList (OrdInt)

# IntOrdList.empty
  |> IntOrdList.add 5
  |> IntOrdList.add 3
  |> IntOrdList.add 10
  |> IntOrdList.to_list
- : IntOrdList.elem list = [3; 5; 10]
```

Useful Library Types, Modules and Functors

- Option types: `type 'a option = None | Some of 'a`
- Set modules: obtained through the functor `Set.Make` (requires a module OrderedType)

```
module ISet = Set.Make(Int)

ISet.union (ISet.singleton 0) (ISet.singleton 2)
```

- Map modules: obtained through the functor `Map.Make` (requires a module OrderedType)

```
module SMap = Map.Make(String)
let m = SMap.add "ciao" 0 (SMap.singleton "boh" 0)

# SMap.find_opt "ciao" m;;
- : int option = Some 0

# SMap.find_opt "bao" m;;
- : int option = None
```

Exceptions

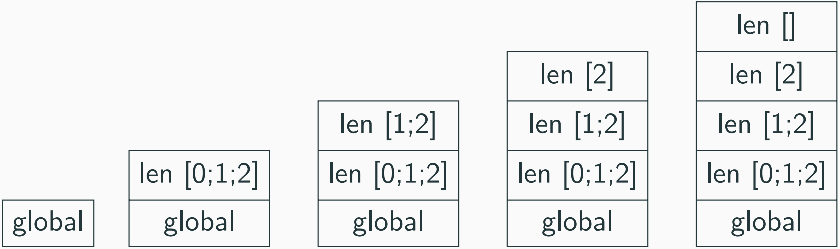
```
5 / 0;;
Exception: Division_by_zero.

# try 5 / 0 with Division_by_zero -> 42;;
- : int = 42

# exception My_exception of string;;
exception My_exception of string

# try if true then raise (My_exception "hello") else 0
  with My_exception s -> print_endline s; 42;;
hello
- : int = 42
```

Tail Recursion

How to manage function call in the compiler?

- Names and associated values are local to the function
- It starts with the global environment and create an activation record for each call
- Activation records are stacked: names are resolved as locally as possible
- Using recursion instead of while loops can cause problem when activation records are
- The famous stack overflow error!

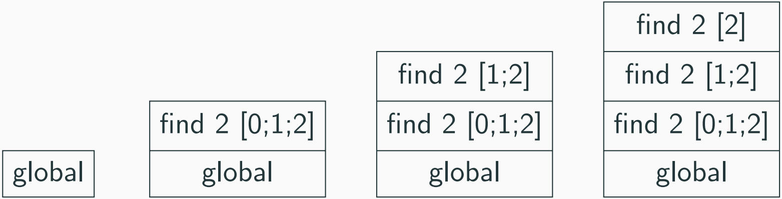
```
let rec len ls = match ls with
  | [] -> 0
  | _::ls -> 1 + (len ls)
  in lrev [0;1;2]
```



In the end: `1 + (1 + (1 + 0)) = 3`

Tail Recursion : the recursive value is just returned!

```
let rec find x ls = match ls with
  | [] -> false
  | l::_ when l = x -> true
  | _::ls' -> find x ls'
  in find 2 [0;1;2]
```



For each frame: a value for `ls`, for `l`, and return value

In the end: just `true`! **No need to keep the records!**

Tail Recursion : the recursive value is just returned!

```
let rec find x ls = match ls with
  | [] -> false
  | l::_ when l = x -> true
  | _::ls' -> find x ls'
  in find 2 [0;1;2]
```

The compiler is smart enough to compile it into the following:

A lot more efficient (both in space and time)!

Note: library functions are tail recursive (see the documentation)

E.g. `fold_left` is tail recursive, `fold_right` is not!

Tail Recursion though accumulators

A tail recursive len:

```
let len ls =
  let rec helper ls' n =
    match ls' with
    | [] -> n
    | _::ls'' -> helper ls'' (n + 1)
  in helper ls 0
```

Note: we are operating on the parameter, not on the value returned by the function

Tail Recursion in Continuation-Passing-Style

Another tail recursive len:

```
let len ls =
  let rec helper ls' k =
    match ls' with
    | [] -> k 0
    | _::ls'' -> helper ls'' (fun x -> k x + 1)
  in helper ls (fun x -> x)
```

- The function `k` represent how to continue the computation after the recursive call
- A lot less clear then the previous one
- Sometimes it is needed with recursive data structures that are not linear (e.g. trees)

Exercises

Exercise 8. Define a type for Finite State Automata and a function for checking if a given string is inside the generated language. But this time use Maps and Sets!

Exercise 9. Write a tail recursive function for computing the sum of the elements in the leaves of a binary tree of integers.

Exercise 10. Write a module with an abstract type `stack` and with functions for pushing a value on top of the stack (`push n`), one for removing and returning the element on top (`pop`), and one for updating the two top elements by performing an arithmetical operation (`update`). Define a type for programs composed by sequences of pushes and updates, and a function `exec` to get the result of executing it over the empty stack.

Example: `exec [Push 2; Push 3; Push 4; Apply sum]` returns a stack with 7 on top and then 2.