# Functional Programming in OCaml

Kihong Heo

KAIST

# Interacting with OCaml

- In a terminal, type `utop` to start the REFL (read-eval-print-loop)
- An expression with `;;` in the REFL tells you the resulting **value** and its **type**

```
$ utop
# 42;;
- : int = 42
# 1 + 1 (* this is a comment *);;
- : int 2
```

- Use the OCaml compiler (`ocamlc`) to compile and execute programs

```
$ ocamlc -o hello hello.ml
$ ./hello
Hello, World!
```

# Characteristics

- Statically-typed, strongly-typed language
  - Compile-time type checking and no type errors at run time (guaranteed)
  - Each value and expression has a type and no implicit type casting
- Garbage-collected language
  - The garbage collector automatically performs memory management
- Multi-paradigm language (but **some** are more preferred than **others**)
  - value / object, immutable / mutable, pure / side-effect

# Values, Names, Expressions, and Definitions

# Values

- OCaml is so called a “value-oriented” language
  - value: immutable, object: mutable
- The ultimate goal of OCaml programs: compute values
  - Programs **generate** new values using old values
  - **NOT changing** old objects to new object

# Expressions

- The primary piece of OCaml
  - Imperative programs: built out of commands
  - Functional programs: built out of expressions
- The primary task of computation: to evaluate an expression to a value
- Example: true, `2+2`, `increment 21`
- An expression evaluates to value, raises an exception, or never terminates

# Arithmetic Expressions

- Arithmetic expressions evaluate to numbers
- OCaml has more or less all the usual arithmetic operators in other languages

```
# 1 + 2 - 3;;
- : int = 0
```

- Arithmetic operators on integers:

```
# 1 + 2      (* addition *)
# 1 - 2      (* subtraction *)
# 1 * 2      (* multiplication *)
# 1 / 2      (* division *)
# 1 mod 2    (* modulo *)
```

- Arithmetic operators on floating point numbers:

```
# 1.1 +. 2.1 (* addition *)
# 1.1 -. 2.1 (* subtraction *)
# 1.1 *. 2.1 (* multiplication *)
# 1.1 /. 2.1 (* division *)
```

# Boolean Expressions

- Boolean expressions evaluate to boolean values
- OCaml has more or less all the usual boolean operators in other languages

```
# true;;
- : bool = true
# false;;
- : bool = false
# 1 + 1 > 2;;
- : bool = false
```

- Boolean operators

```
# x = y     (* equal *)
# x <> y    (* not equal *)
# x < y     (* less than *)
# x <= y    (* less than or equal *)
# x > y     (* larger than *)
# x >= y    (* larger than or equal *)
# x && y    (* logical and *)
# X || y    (* logical or *)
```

# Character and String Expressions

- Character expressions evaluate to character values

```
# ‘a’;;
- : char = ‘a’
# ‘a’ = ‘a’;;
- : bool = true
```

- String expressions evaluate to string values (primitive values in OCaml)

```
# “hello”;;
- : string = “hello"

# “hello” = “hello”;;
- : bool = true

# “hello, “ ^ ”world!”;;
- : string = “hello, world!”
```

# If Expressions

- If-then-else is an expression, not a statement as in imperative languages
- `if e1 then e2 else e3` evaluates to
    - e2 if e1 evaluates to true
    - e3 if e1 evaluates to false
    - e1 : bool, e2 : t, e3 : t, then the whole expression has type t
- Example:

```
# 4 + (if 1 > 2 then 1 else 2);;
- : int = 6
```

# Definitions

- Definitions are not expressions (i.e., not evaluate to values)
- Definitions bind values to names

```
let x = 42
```

- An OCaml program is a sequence of definitions (no need for `main`)

```
let x = “Hello, World!”
let _ = print_endline x
```

# Function Definitions

- Function definition

```
# let increment x = x + 1;;
val increment : int -> int = <fun>
# increment 0;;
- : int = 1
```

- Recursive function definition using `let rec`:

```
# let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)
val fact : int -> int = <fun>
```

```
let rec pow x y =
  if y = 0 then 1 else x * pow (y - 1)
val pow : int -> int = <fun>
```

# Anonymous Function

- Not all values have to have names

```
# 42;;
- : int = 42
```

- Anonymous function value (so called lambda)

```
# fun x -> x + 1;;
- : int -> int = <fun>
```

- The followings are semantically equivalent

```
let inc x = x + 1
let inc = fun x -> x + 1
```

# Function Application

- Application-style: e0 e1 e2 … en
  where e0 is a function and e1 through en are the arguments

```
square (inc 5)
```

- Pipeline-style: “values are sent through the pipeline from left to right”

```
5 |> inc |> square
```

# Polymorphic Functions

- Functions that take many types of values
- Example: the identity function

```
# let id x = x;;
id : ‘a -> ‘a = <fun>
```

- ‘a, ‘b, … are type variables that stand for unknown type

# Partial Application

- A function that is partially applied to a subset of the arguments

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

# let add5 = add 5
val add5 : int -> int = <fun>

# add5 2;;
- : int = 7
```

# Higher-order Functions

- **Functions** that take other **functions** as arguments or return **functions**
  - “Functional”!
  - because functions are first-class values like integers

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

# let add5 = add 5
val add5 : int -> int = <fun>

# let add_high f x = (f (x + 1)) + 2;;
val add_high : (int -> int) -> int -> int = <fun>

# add_high add5 10;;
- : int = 17

# add_high (fun x -> x + 5) 10;;
- : int = 17
```

# Let-in Expressions

- Expressions with local name binding with `let`

```
# let x = 42 in x + 1
- : int = 43
```

- Let bindings are in effect only in the block of code in which they occur

```
let x = 42 in
  (* y is not meaningful here *)
  x
  +
  (let y = "3110" in
    (* y is meaningful here *)
    int_of_string y)
```

# Pattern Matching

- A control structure for elegant case analysis (`match` … `with` …)

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1)
```

```
let rec fibo n =
  if n = 0 then 1
  else if n = 1 then 1
  else fibo(n-1) + fibo(n-2)
```

```
let rec fibo n =
  match n with
  | 0 | 1 -> 1
  | _ -> fibo(n - 1) + fibo(n - 2)
```

# Pattern Matching with Function

- Simplified definitions with `function`

```
let rec fact = function
  | 0 -> 1
  | _ -> n * fact (n - 1)
```

```
let rec fibo = function
  | 0 | 1 -> 1
  | _ -> fibo(n - 1) + fibo(n - 2)
```

# Data Types and Pattern Matching

# Lists

- A sequence of values all of which have the same type

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [true; false; false];;
- : bool list = [true; false; false]
# [[1;2;3];[4;5]];;
- : int list list = [[1;2;3];[4;5]];;
```

- In principle, there are only two ways to build a list: nil ([]) and cons (::)

```
# [] (* nil means the empty list*) ;;
- : ‘a list
# 1::[2; 3] (* cons prepends an elem to a list *);;
- : int list [1; 2; 3]
# 1::2::3::[] (* conceptually the same as above *);;
- : int list [1; 2; 3]
```

# Pattern Matching with Lists

- Use pattern matching when doing a case analysis for lists
  - Remember that there are only two cases: nil and cons (head and tail)

```
(* compute the sum of an int list *)
let rec sum lst =
  match lst with
  | [] -> 0
  | h::t -> h + sum t
```

```
(* compute the length of a list *)
let rec length lst =
  match lst with
  | [] -> 0
  (* don't give a name to the head *)
  | _::t -> 1 + length t
```

# Tuples

- An ordered collection of (possibly different types of) values
- A kind of *product* type: a Cartesian product of multiple sets

```
# (1, 2, 10);;
- : int * int * int = (1, 2, 10)

# (true, “Hello”);;
- : bool * string = (true, "Hello")

# (true, “Hello”, (1, 2));;
- : bool * string * (int * int) = (true, “Hello”, (1, 2))
```

# Pattern Matching with Tuples

- Examples:

```
(* OK *)
let thrd t =
  match t with
  | (x, y, z) -> z

(* good *)
let thrd t =
  let (x, y, z) = t in z

(* better *)
let thrd t =
  let (_, _, z) = t in z

(* best *)
let thrd (_, _, z) = z
```

```
let logical_and x y =
  match (x, y) with
  | (true, true) -> true
  | (_, _) -> false

let logical_or x y =
  match (x, y) with
  | (true, _)
  | (_, true) -> true
  | (_, _) -> false
```

# Records

- A composite of other types of data, each of which named
  - elements of a tuple: identified by position
  - elements of a record: identified by name

```
(* a record type definition *)
type student = { name: string; sid: int }
```

- Record expressions evaluate to record value

```
# let me = { name = “Kihong”; sid = 2020 };;
val me : student = {name = "Kihong"; sid = 2020}
```

- The dot expression gets a field from a record

```
# me.sid;;
- : int = 2020
```

# Pattern Matching with Records

```
(* OK *)
let get_sid s =
  match m with
  | { name=n; sid=s } -> s

(* good *)
let get_sid s =
  match m with
  | { name=_; sid=s } -> s

(* better *)
let get_sid s =
  match s with
  | { sid } -> sid

(* best *)
let get_sid s = s.sid
```

# Type Synonyms

- A new name for an already existing type
  - similar to `typedef` in C

```
type point = float * float
type vector = float list
type matrix = float list list

let get_x (x, _) = x
```

# Options

- A value representing an optional value
  - A value with type t in some cases or nothing for the rest
  - Built by using `Some` and `None`

```
# Some 42;;
- : int option = Some 42

# None;;
- : ‘a option
```

```
let rec list_max lst =
  match lst with
  | []   -> None
  | h::t -> (match list_max t with
             | None -> Some h
             | Some m -> Some (max h m))
```

# Pattern Matching with Options

- Example: a function that extracts an integer from an option value

```
let extract o =
  match o with
  | Some i  -> string_of_int i
  | None -> “”
```

# Variants (1)

- A data type representing a value that is one of several possibilities
  - Similar to enum in C or Java but more powerful
  - A kind of *sum* type: union of multiple sets
  - Individual names of the values of a variant are called **constructors**

```
# type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat;;
- type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

# let d = Tue;;
val d : day = Tue
```

# Pattern Matching with Variants (1)

```
let int_of_day d =
  match d with
  | Sun -> 1
  | Mon -> 2
  | Tue -> 3
  | Wed -> 4
  | Thr -> 5
  | Fri -> 6
  | Sat -> 7

let good_day d =
  match m with
  | Sun | Sat -> true
  | _ -> false
```

# Variants (2)

- Each constructor can have arguments
  - more than just enumerating finite set of elements (which is a special case: one value per a constructor)
  - carrying additional data

```
type shape =
   | Point  of point
   | Circle of point * float   (* center and radius *)
   | Rect   of point * point   (* lower-eft and
                                  upper-right corners *)

let rect1 = Rect (1.0, 5.7)
let circle1 = Circle ((0.6, 0.3), 50.5)
```

# Pattern Matching with Variants (2)

```
type shape =
  | Point  of point
  | Circle of point * float  (* center and radius *)
  | Rect   of point * point  (* lower-eft and
                                upper-right corners *)

let area s =
  match s with
  | Point _ -> 0.0
  | Circle (_, r) -> pi *. (r ** 2.0)   (* ** means power *)
  | Rect ((x1, y1), (x2, y2)) ->
      let w = x2 -. x1 in
      let h = y2 -. y1 in
      w *. h
```

# Variants (3)

- Recursive variants are also possible
  - Variants mentioning their own name inside their own body

```
type intlist = Nil | Cons of int * intlist

let lst3 = Cons (3, Nil)                (* conceptually, 3::[] *)
let lst123 = Cons (1, Cons (2, lst3)) (* conceptually, [1; 2;,3] *)
```

# Unit

- A value and a type that mean nothing
  - Similar to void in C and Java

```
# ();;
- : unit = ()
```

- Mostly used for side-effect only expressions such as print

```
# print_endline;;
- : string -> unit = <fun>
# print_endline “Hi”;;
Hi
- : unit = ()
```

# Exceptions

- Conventional exception mechanism similar to other languages
- A new type of exception is defined with `exception`

```
exception Division_by_zero
exception Failure of string

let rec div x y =
  if y = 0 then raise Division_by_zero
  else x / y
```

# Pattern Matching with Exceptions

- Pattern matching for exceptions with `try`
  - Try to evaluate an expression and handle each exception if raised

```
let r =
  try div x y with
  | Division_by_zero -> 0
  | Not_implemented -> -1
```

- Pattern matching for exceptions with `match`

```
let r =
  match div x y with
  | n -> string_of_int n
  | Division_by_zero -> “Division_by_zero”
  | Not_implemented -> “Not_implement”
```

# Modules

# Module

- A set of definitions of types, values, and exceptions
  - Similar to class in Java and C++ (except objects)

```
module ListStack = struct
  type ‘a stack = ‘a list
  exception Empty

  let empty = []
  let is_empty s = (s = [])
  let push x s = x :: s
  let peek = function
    | [] -> raise Empty
    | x::_ -> x
  let pop = function
    | [] -> raise Empty
    | _::xs -> xs
end
```

# Module Type

- A set of declarations of types, values, and exceptions
  - Similar to interface in Java

```
(* all capital by convection *)
module type STACK = sig
  type ‘a stack
  exception Empty
  val empty : ‘a stack
  val is_empty : ‘a stack -> bool
  val push : ‘a -> ‘a stack -> ‘a stack
  val peek : ‘a stack -> ‘a
  val pop : ‘a stack -> ‘a stack
end

module ListStack : STACK = struct
  (* the rest is the same as before *)
end
```

# Functors

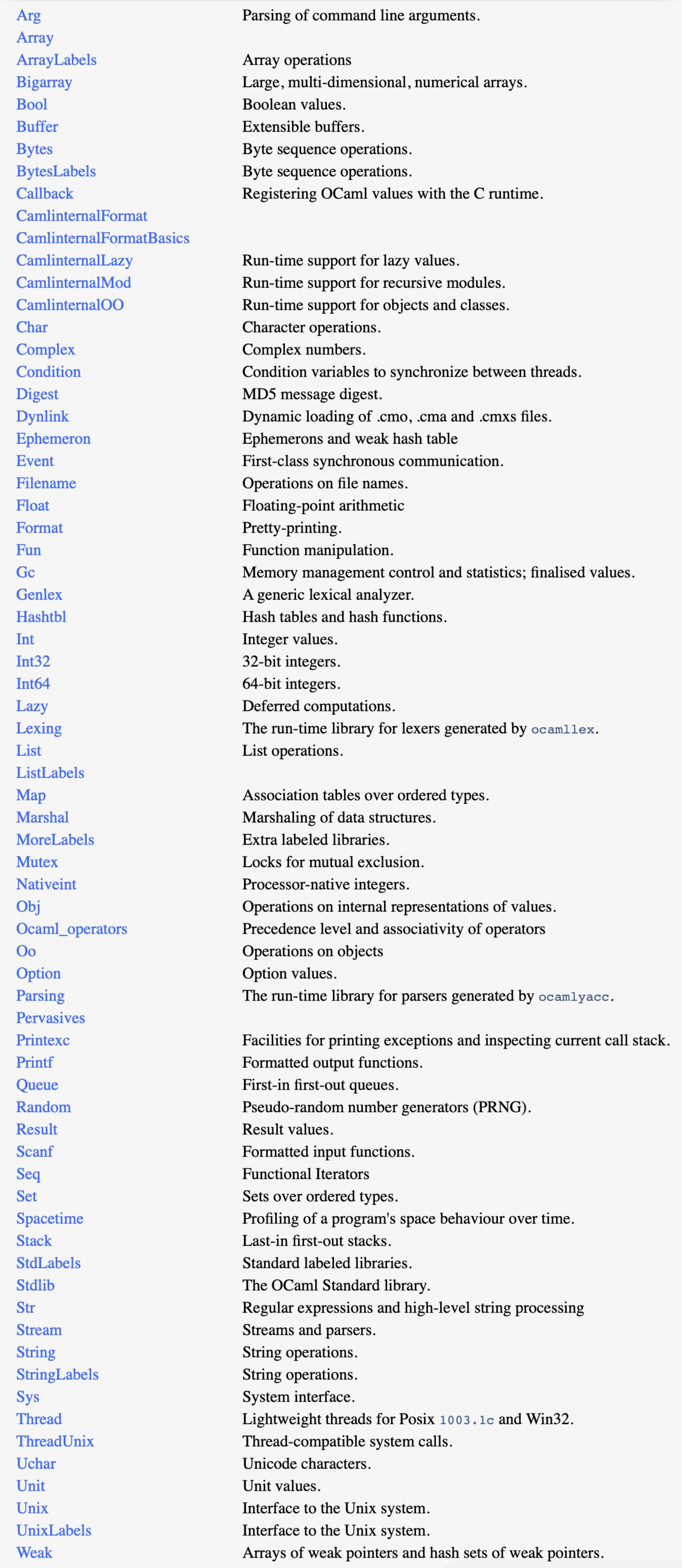
- A function from modules to modules
  - Similar to template (C++) and generic (Java)

```
module type X = sig
  val x : int
end

module IncX (M: X) = struct
  let x = M.x + 1
end

module A = struct let x = 0 end
(* A.x is 0 *)

module B = IncX(A)
(* B.x is 1 *)
```

# Libraries

# OCaml Standard Libraries

- A collection of many useful modules
- Data structures, algorithms, system calls, etc
- http://caml.inria.fr/pub/docs/manual-ocaml/libref
- Also, an awesome reference for OCaml programming
  - https://github.com/ocaml/ocaml/tree/trunk/stdlib



| | |
|---|---|
| Arg | Parsing of command line arguments. |
| Array | |
| ArrayLabels | Array operations |
| Bigarray | Large, multi-dimensional, numerical arrays. |
| Bool | Boolean values. |
| Buffer | Extensible buffers. |
| Bytes | Byte sequence operations. |
| BytesLabels | Byte sequence operations. |
| Callback | Registering OCaml values with the C runtime. |
| CamlinternalFormat | |
| CamlinternalFormatBasics | |
| CamlinternalLazy | Run-time support for lazy values. |
| CamlinternalMod | Run-time support for recursive modules. |
| CamlinternalOO | Run-time support for objects and classes. |
| Char | Character operations. |
| Complex | Complex numbers. |
| Condition | Condition variables to synchronize between threads. |
| Digest | MD5 message digest. |
| Dynlink | Dynamic loading of .cmo, .cma and .cmxs files. |
| Ephemeron | Ephemerons and weak hash table |
| Event | First-class synchronous communication. |
| Filename | Operations on file names. |
| Float | Floating-point arithmetic |
| Format | Pretty-printing. |
| Fun | Function manipulation. |
| Gc | Memory management control and statistics; finalised values. |
| Genlex | A generic lexical analyzer. |
| Hashtbl | Hash tables and hash functions. |
| Int | Integer values. |
| Int32 | 32-bit integers. |
| Int64 | 64-bit integers. |
| Lazy | Deferred computations. |
| Lexing | The run-time library for lexers generated by `ocamllex`. |
| List | List operations. |
| ListLabels | |
| Map | Association tables over ordered types. |
| Marshal | Marshaling of data structures. |
| MoreLabels | Extra labeled libraries. |
| Mutex | Locks for mutual exclusion. |
| Nativeint | Processor-native integers. |
| Obj | Operations on internal representations of values. |
| Ocaml_operators | Precedence level and associativity of operators |
| Oo | Operations on objects |
| Option | Option values. |
| Parsing | The run-time library for parsers generated by `ocamlyacc`. |
| Pervasives | |
| Printexc | Facilities for printing exceptions and inspecting current call stack. |
| Printf | Formatted output functions. |
| Queue | First-in first-out queues. |
| Random | Pseudo-random number generators (PRNG). |
| Result | Result values. |
| Scanf | Formatted input functions. |
| Seq | Functional Iterators |
| Set | Sets over ordered types. |
| Spacetime | Profiling of a program's space behaviour over time. |
| Stack | Last-in first-out stacks. |
| StdLabels | Standard labeled libraries. |
| Stdlib | The OCaml Standard library. |
| Str | Regular expressions and high-level string processing |
| Stream | Streams and parsers. |
| String | String operations. |
| StringLabels | String operations. |
| Sys | System interface. |
| Thread | Lightweight threads for Posix `1003.1c` and Win32. |
| ThreadUnix | Thread-compatible system calls. |
| Uchar | Unicode characters. |
| Unit | Unit values. |
| Unix | Interface to the Unix system. |
| UnixLabels | Interface to the Unix system. |
| Weak | Arrays of weak pointers and hash sets of weak pointers. |

# Stdlib

- The `Stdlib` module is automatically opened
  - Specifying module name (`Stdlib`) is not needed
- A lot of basic operations over the built-in types (numbers, booleans, strings, I/O channels, etc)
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

# List (1)

- The List module has a number of utility functions for lists
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
- Builders: constructing a new list

```
(* val cons : 'a -> 'a list -> 'a list *)
let lst = [1;2;3]
let x = List.cons 1 lst
let y = 1::lst (* simpler form *)

(* val append : 'a list -> 'a list -> 'a list
let z = List.append x y
let w = x @ y (* simpler form *)
```

# List (2)

- Iterators: iterating over lists

```
(* val iter : ('a -> unit) -> 'a list -> unit *)
let l = [1;2;3]
let _ = List.iter print_int lst

(* val map : ('a -> 'b) -> 'a list -> 'b list *)
let str_lst = List.map string_of_int lst
(* str_lst = ["1"; "2"; "3"] *)

(* val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list *)
let sum = List.fold_left (+) 0 lst
let sum_of_squre = List.fold_left (fun acc x -> acc + x * x) 0 lst
```

# List (3)

- Searching

```
(* find_opt : ('a -> bool) -> 'a list -> 'a option *)
let one = List.find_opt (fun x -> x > 1) lst

(* filter : ('a -> bool) -> 'a list -> 'a list *)
let larger_than_one = List.filter (fun x -> x > 1) lst
```

- All together with pipelining

```
let r = (* define r by doing the following *)
  1::lst                               (* prepend 1 to lst *)
  |> List.map (fun x -> x * x)         (* apply square for all the elems *)
  |> List.filter (fun x -> x > 1)      (* get all the elems larger than 1 *)
  |> List.fold_left ( + ) 0            (* compute the sum of all the elems *)
```

# Set (1)

- The set data structure and functions
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html
- The `Set.Make` functor constructs implementations for any type
  - The argument module must have a type `t` and a `compare` function

```
module IntPairs = struct
  type t = int * int
  (* a total ordering function with type t -> t -> int is required *)
  (* compare x y is -1 if x < y, 0 if x = y, 1 otherwise *)
  let compare (x0, y0) (x1, y1) =
    match compare x0 x1 with  (* this compare is a builtin function *)
    | 0 -> compare y0 y1
    | c -> c
end
module PairSet = Set.Make(IntPairs)
```

# Set (2)

- Builders

```
(* type elt is the type of the set element *)

(* val empty : t *)
let emptyset = PairSet.empty

(* val add : elt -> t -> t *)
let x = PairSet.add (1,2) emptyset

(* val singleton : elt -> t *)
let y = PairSet.singleton (1,2)

(* val remove : elt -> t -> t *)
let z = PairSet.remove (1,2) y

(* set operators with type t -> t -> t *)
let u = PairSet.union x y
let i = PairSet.inter x y
let d = PairSet.diff x y
```

# Set (3)

- Iterators

```
(* val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a *)
let sum_left = PairSet.fold (fun (i, _) s -> i + s) x 0

(* val iter : (elt -> unit) t -> t *)
let _ = PairSet.iter (fun i, _) -> print_int i) x

(* val map : (elt -> elt) -> t *)
let double = PairSet.map (fun (i, j) -> (2 * i, 2 * j)) x
```

- Searching

```
(* val mem : elt -> t -> bool *)
let membership = PairSet.mem (1, 2) x

(* val filter : (elt -> bool) -> t -> t *)
let big_left = PairSet.filter (fun (i, j) -> i > j) x
```

# Map (1)

- The map data structure (kay-value pairs) and functions
  - [http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html](http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html)
- The `Map.Make` functor constructs implementations for any type of key
  - The argument module must have a type `t` and a `compare` function

```
module IntPairs = struct
  type t = int * int
  (* a total ordering function with type t -> t -> int is required *)
  (* compare x y is -1 if x < y, 0 if x = y, 1 otherwise *)
  let compare (x0, y0) (x1, y1) =
    match compare x0 x1 with  (* this compare is a builtin function *)
    | 0 -> compare y0 y1
    | c -> c
end
module PairMap = Map.Make(IntPairs)
```

# Map (2)

- Builders

```
(* type key is the type of the map keys *)
(* type ‘a t is the type of maps is from type key to type ‘a *)

(* val empty : ’a t *)
let emptymap = PairMap.empty

(* val add : key -> ‘a -> bool *)
let x = PairMap.add (1,2) “one-two” emptymap

(* val singleton : key -> ‘a -> ’a t *)
let y = PairMap.singleton (1,2) “one-two”

(* val remove : key -> ’a t -> ’a t *)
let z = PairMap.remove (1,2) y
```

# Map (3)

- Iterators

```
(* val fold : (key -> ‘a -> ‘b -> ‘b) -> ’a t -> ‘b -> ‘b *)
let sum = PairMap.fold (fun (i, j) _ s -> (i + j) x 0

(* val iter : (elt -> unit) t -> t *)
let _ = PairMap.iter (fun (i, _) -> print_int i) x

(* val map : ('a -> ‘b) -> 'a t -> ‘b t *)
let double = PairMap.map (fun str -> String.length str) x
```

- Searching

```
(* val mem : key -> ’a t -> bool *)
let membership = PairMap.mem (1, 2) x

(* val filter : (key -> ‘a -> bool) -> ‘a t -> ‘a t *)
let big_left = PairMap.filter (fun (i, j) _ -> i > j) x
```


Get Into The Wild Adventure

# Reference: Books and Tutorials

- Real World OCaml: http://dev.realworldocaml.org
- Functional Programming in OCaml: https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook
- OCaml tutorials: https://ocaml.org/learn/tutorials/

# Reference: Real World OCaml Code

- Learn more from real-world OCaml code!
  - OCaml compiler: https://github.com/ocaml/ocaml
  - Sparrow: https://github.com/prosyslab/sparrow
  - Infer: https://github.com/facebook/infer
  - OCamlgraph: https://github.com/backtracking/ocamlgraph