Syntax
Implementations are in .ml files, interfaces are in .sig files.
Comments can be nested, between delimiters (*...*)
Integers: 123, 1_234_567_890
Chars: ‘a’, ‘255’, ‘\xFF’, ‘\n’

Data Types
- unit: void, takes only one value: ()
- int: integer of either 31 or 63 bits, like 42
- int32: 32 bits Integer, like 42
- int64: 64 bits Integer, like 42L
- float: double precision float, like 1.0
- bool: boolean, takes two values: true or false
- char: simple ASCII characters, like ‘a’
- string: strings, like “Hello” or foo|Hello|foo
- array: mutable string of chars
  - `a list`: lists, like head :: tail or [1;2;3]
  - `a array`: arrays, like [[1;2;3]]
  - `t n`: tuple, like (1, “foo”, ‘b’)

Constructed Types
- type record = new record type
  { field1 : bool; immutable field
    mutable field2 : int; } mutable field
- type enum = new variant type
  | Constant | Constant constructor
  | Param of string | Constructor with arg
  | Pair of string * int | Constructor with args
  | Gadt : int -> enum | GADT constructor
  | Inline of { x : int } | Inline record

Constructured Values
- let r = { field1 = true; field2 = 3; }
- let r’ = { r with field1 = false }
- r.field2 <- r.field2 + 1;
- let c = Constant
- let c = Param “foo”
- let c = Pair (“bar”,3)
- let c = Gadt 0
- let c = Inline { x = 3 }

References, Strings and Arrays
- let x = ref 3 integer reference (mutable)
- x := 4 reference assignment
- print_int !x; reference access
- s.[0] string char access
- t.(0) array element access
- t.(0) <- x array element modification

Imports — Namespaces
- open Unix global open
- let open Unix in expr local open
- Unix.(expr) local open

Functions
- let f x = expr function with one arg
- func f x = expr recursive function
- let f x y = expr apply: f x with two args
- let f (x,y) = expr apply: f x y with a pair as arg
- List.iter (fun x -> expr) l anonymous function
- let fn = function None -> act function definition
  | Some x -> act [by cases]
- let f ->str-len = expr apply (for ->str-str): f ->str ->len
- let f ?len ->str = expr apply with optional arg
- let f ?len = expr apply with omitted arg
- let f ?len = expr optional arg default
- let f ?len = expr apply with client option
- let f (x : int) = expr arg has constrained type
- let f : a.b.* b -> a function with constrained type

Modules
- module M = struct ... end module definition
- module M = Unix module and signature
- module M = Sig module type of M
- module M = struct ... end module type
- let m = (module M : Sig) module
- module Make(S: Sg) = struct module
  | Include M include
  | virtual foo x = virtual methord
  | is a subtype
  | inheritance and ancestor reference
  | superclass
  | virtual function

Pattern-matching
- match expr with
  | pattern -> action
  | pattern when guard -> action conditional case
  | _ -> action
default case

Patterns:
- | Pair (x,y) -> variant pattern
- | (field = 3 ; _ ) -> record pattern
- | head :: tail -> list pattern
- | [1;2;x] -> list pattern
- | (Some x) as y -> or-pattern
- | (x,0) -> or-pattern
- | exception ezn -> try&match

References
- let x = ref 3 integer reference (mutable)
- x := 4 reference assignment
- print_int !x; reference access
- s.[0] string char access
- t.(0) array element access
- t.(0) <- x array element modification

Imports — Namespaces
- open Unix global open
- let open Unix in expr local open
- Unix.(expr) local open

Conditions
Do NOT use on closures
- Structural | Physical
  | <= | == |
  Polyorphic Equality | Polyorphic Inequality

Polyorphic Generic Comparison Function: compare
- x < y
- x = y
- x > y

Loops
- while cond do ... done;
- for var = min_value to max_value do ... done;
- for var = max_value downto min_value do ... done;

Exceptions
- exception MyExn new exception
- exception MyExn of t * t’ new exception with args
- raise (MyExn (args)) raise an exception
- try expr try expr
- with MyExn -> ... catch MyExn

Objects and Classes
- class virtual foo x = virtual class with arg
- let y = x+2 in
- object (self : ‘a) init before object creation
- with self reference
- mutable instance variable
- accessor
- method get = variable
- method set z = variable <- z+y
- method virtual copy : ‘a virtual method
- init after object creation
- with MyExn -> ... catch MyExn

Polymorphic variants
- type t = [ ‘A | ‘B of int ] type t = [ ‘A | ‘C of float ]
- closed variant
- type u = [ ‘A | ‘C of float ]
- union of variants
- type v = [ t | u | ]
- argument must be
- of the argument