



## Syntax

Implementations are in `.ml` files, interfaces are in `.mli` files.  
 Comments can be nested, between delimiters `(*...*)`  
 Integers: 123, 1\_000, 0x4533, 0o773, 0b1010101  
 Chars: 'a', '\255', '\xFF', '\n'    Floats: 0.1, -1.234e-34

## Data Types

`unit`        Void, takes only one value: `()`  
`int`         Integer of either 31 or 63 bits, like 32  
`int32`      32 bits Integer, like 321  
`int64`      64 bits Integer, like 32L  
`float`      Double precision float, like 1.0  
`bool`       Boolean, takes two values: `true` or `false`  
`char`       Simple ASCII characters, like 'A'  
`string`     Strings of chars, like "Hello"  
`'a list`    Lists, like `head :: tail` or `[1;2;3]`  
`'a array`   Arrays, like `[|1;2;3|]`  
`t1 * ... * tn` Tuples, 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

## Constructed 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)
```

## 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
s.[0] <- 'a'    string char modification
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
let rec f x = expr            recursive function
                              apply: f x
                              with two args
let f x y = expr              apply: f x y
                              with a pair as arg
let f (x,y) = expr            apply: f (x,y)
                              anonymous function
List.iter (fun x -> e) l      function definition
let f= function None -> act                                    by cases
                              | Some x -> act
                              apply: f (Some x)
let f ~str ~len = expr        with labeled args
                              apply: f ~str:s ~len:10
                              apply (for ~str:str): f ~str ~len
let f ?len ~str = expr        with optional arg (option)
let f ?(len=0) ~str = expr    optional arg default
                              apply (with omitted arg): f ~str:s
                              apply (with commuting): f ~str:s ~len:12
                              apply (len: int option): f ?len ~str:s
                              apply (explicitly omitted): f ?len:None ~str:s
let f (x : int) = expr        arg has constrained type
let f : 'a 'b. 'a*'b -> 'a    function with constrained
                              = fun (x,y) -> x                    polymorphic type
```

## Modules

```
module M = struct .. end        module definition
module M: sig .. end= struct .. end module and signature
module M = Unix                module renaming
include M                      include items from
module type Sg = sig .. end    signature definition
module type Sg = module type of M signature of module
let module M = struct .. end in .. local module
let m = (module M : Sg)        to 1st-class module
module M = (val m : Sg)        from 1st-class module
module Make(S: Sg) = struct .. end functor
module M = Make(M')            functor application
```

Module type items:

`val`, `external`, `type`, `exception`, `module`, `open`, `include`, `class`

## 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 ->                with extra binding
| (1,x) | (x,0) ->                or-pattern
```

## Conditionals

Structural = <>	Physical == !=	Polymorphic Equality Polymorphic Inequality	
-----------------------	----------------------	--	--

Polymorphic Generic Comparison Function: `compare`

	<code>x &lt; y</code>	<code>x = y</code>	<code>x &gt; y</code>
<code>compare x y</code>	-1	0	1

Other Polymorphic Comparisons : `>`, `>=`, `<`, `<=`

## 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'        same with arguments
exception MyFail = Failure        rename exception with args
raise MyExn                        raise an exception
raise (MyExn (args))              raise with args
try expression                    catch MyException if raised
with Myn -> ...                    in expression
```

## Objects and Classes

```
class virtual foo x =              virtual class with arg
  let y = x+2 in                    init before object creation
  object (self: 'a)                object with self reference
    mutable variable = x            mutable instance variable
    method get = variable           accessor
    method set z =                   mutator
      variable <- z+y
    method virtual copy : 'a        virtual method
      initializer                    init after object creation
    end
class bar =                        non-virtual class
  let var = 42 in                  class variable
  fun z -> object                  constructor argument
  inherit foo z as super            inheritance and ancestor reference
  method! set y =                  method explicitly overridden
    super#set (y+4)                access to ancestor
  method copy = {< x = 5 >}        copy with change
  end
let obj = new bar 3                new object
obj#set 4; obj#get                method invocation
let obj = object .. end            immediate object
```

## Polymorphic variants

```
type t = [ 'A | 'B of int ]        closed variant
type u = [ 'A | 'C of float ]
type v = [ t | u | ]                union of variants
let f : [< t ] -> int = function    argument must be
  | 'A -> 0 | 'B n -> n             a subtype of t
let f : [> t ] -> int = function    t is a subtype
  | 'A -> 0 | 'B n -> n | _ -> 1    of the argument
```