

Pervasives

<code>val raise : exn -> 'a</code>	Raise the given exception
<code>val (=) : 'a -> 'a -> bool</code>	tests for structural equality of e1 and e2
<code>val (<>) : 'a -> 'a -> bool</code>	Negation of (=).
<code>val (==) : 'a -> 'a -> bool</code>	e1 == e2 tests for physical equality of e1 and e2.
<code>val (!=) : 'a -> 'a -> bool</code>	Negation of (==).
<code>val (<) : 'a -> 'a -> bool</code>	
<code>val (>) : 'a -> 'a -> bool</code>	
<code>val (<=) : 'a -> 'a -> bool</code>	
<code>val (>=) : 'a -> 'a -> bool</code>	
<code>val compare : 'a -> 'a -> int</code>	compare x y returns 0 if x is equal to y, a negative integer if x is less than y, and a positive integer if x is greater than y.
<code>val min : 'a -> 'a -> 'a</code>	Return the smaller of the two arguments.
<code>val max : 'a -> 'a -> 'a</code>	Return the greater of the two arguments.
<code>val not : bool -> bool</code>	The boolean negation.
<code>val (&&) : bool -> bool -> bool</code>	
<code>val () : bool -> bool -> bool</code>	
<code>val (>) : 'a -> ('a -> 'b) -> 'b</code>	Reverse-application operator: x > f > g is exactly equivalent to g (f (x)).
<code>val (~-) : int -> int</code>	Unary negation.
<code>val (mod) : int -> int -> int</code>	Integer remainder.
<code>val abs : int -> int</code>	Return the absolute value of the argument.
<code>val (**) : float -> float -> float</code>	Exponentiation.
<code>val sqrt : float -> float</code>	Square root.
<code>val log : float -> float</code>	Natural logarithm.
<code>val log10 : float -> float</code>	Base 10 logarithm.
<code>val ceil : float -> float</code>	Round above to an integer value.
<code>val floor : float -> float</code>	Round below to an integer value.
<code>val abs_float : float -> float</code>	abs_float f returns the absolute value of f.
<code>val float_of_int : int -> float</code>	Convert an integer to floating-point.
<code>val int_of_float : float -> int</code>	
<code>val (^) : string -> string -> string</code>	String concatenation.
<code>val int_of_char : char -> int</code>	
<code>val char_of_int : int -> char</code>	
<code>val fst : 'a * 'b -> 'a</code>	Return the first component of a pair.



By [benjenkinsv95](#)

Published 9th March, 2017.

Last updated 9th March, 2017.

Page 1 of 5.

Sponsored by [Readable.com](#)

Measure your website readability!

<https://readable.com>

Pervasives (cont)

<code>val snd : 'a * 'b -> 'b</code>	Return the second component of a pair.
<code>val (@) : 'a list -> 'a list -> 'a list</code>	List concatenation.

Types and Type Inference

Function	Type
<code>fun x y -> x + y</code>	<code>int -> int -> int</code>

Include variables that are passed in inside of the type.

List basics

<code>val length : 'a list -> int</code>	Return the length
<code>val nth : 'a list -> int -> 'a</code>	Return the n-th element of the given list.
<code>val rev : 'a list -> 'a list</code>	List reversal.
<code>val concat : 'a list list -> 'a list</code>	Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result.

Prefix with `List.`

List Iterators

<code>val map : ('a -> 'b) -> 'a list -> 'b list</code>	
<code>val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list</code>	Same as <code>List.map</code> , but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.
<code>val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a</code>	
<code>val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b</code>	
<code>val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list</code>	
<code>val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a</code>	
<code>val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c</code>	

Prefix with `List.`

List Scanning

<code>val for_all : ('a -> bool) -> 'a list -> bool</code>	checks if all elements of the list satisfy the predicate <code>p</code>
<code>val exists : ('a -> bool) -> 'a list -> bool</code>	checks if at least one element of the list satisfies the predicate <code>p</code>
<code>val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool</code>	
<code>val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool</code>	



By [benjenkinsv95](#)

Published 9th March, 2017.
Last updated 9th March, 2017.
Page 2 of 5.

Sponsored by [Readable.com](#)
Measure your website readability!
<https://readable.com>

List Scanning (cont)

`val mem : 'a -> 'a list -> bool` mem a l is true if and only if a is equal to an element of l.

`val memq : 'a -> 'a list -> bool` Same as List.mem, but uses physical equality

Prefix with `List.`

List Searching

`val find : ('a -> bool) -> 'a list -> 'a` returns the first element of the list l that satisfies the predicate p

`val filter : ('a -> bool) -> 'a list -> 'a list` returns all the elements of the list l that satisfy the predicate p

`val partition : ('a -> bool) -> 'a list -> 'a list * 'a list` returns a pair of lists (l1, l2), where l1 is the list of all the elements of l that satisfy the predicate p, and l2 is the list of all the elements of l that do not satisfy p

Prefix with `List.`

Lists of pairs

`val split : ('a 'b) list -> 'a list * 'b list` Transform a list of pairs into a pair of lists

`val combine : 'a list -> 'b list -> ('a * 'b) list` Transform a pair of lists into a list of pairs

Prefix with `List.`

List Sorting

`val sort : ('a -> 'a -> int) -> 'a list -> 'a list`

`val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list` Merge two lists: Assuming that l1 and l2 are sorted according to the comparison function cmp, merge cmp l1 l2 will return a sorted list containing all the elements of l1 and l2.

Prefix with `List.`

String basics

`val length : string -> int`

`val get : string -> int -> char` returns the character at index n

`val make : int -> char -> string` String.make n c returns a fresh string of length n, filled with the character c.

`val sub : string -> int -> int -> string` returns a fresh string of length len, containing the substring of s that starts at position start and has length len.



By [benjenkinsv95](#)

Published 9th March, 2017.

Last updated 9th March, 2017.

Page 3 of 5.

Sponsored by [Readable.com](#)

Measure your website readability!

<https://readable.com>

String basics (cont)

<code>val concat : string -> string list -> string</code>	String.concat sep sl concatenates the list of strings sl, inserting the separator string sep between each.
<code>val map : (char -> char) -> string -> string</code>	String.map f s applies function f in turn to all the characters of s (in increasing index order) and stores the results in a new string that is returned.
<code>val trim : string -> string</code>	
<code>val index : string -> char -> int</code>	returns the index of the first occurrence of character c in string s
<code>val rindex : string -> char -> int</code>	returns the index of the last occurrence of character c in string s
<code>val contains : string -> char -> bool</code>	
<code>val uppercase_ascii : string -> string</code>	all lowercase letters translated to uppercase
<code>val lowercase_ascii : string -> string</code>	all uppercase letters translated to lowercase
<code>val capitalize_ascii : string -> string</code>	first character set to uppercase
<code>val uncapitalize_ascii : string -> string</code>	first character set to lowercase

Prefix with `String.`

Map

```
let rec map ( f : 'a -> 'b) (lst : 'a list) : 'b list =
  match lst with
  | [] -> []
  | hd :: tl -> f hd :: map f tl ;;
```

fold_left

```
let rec foldleft (f : 'a -> 'b -> 'a)
  (acc : 'a)
  (lst : 'b list)
  : 'a =
  match lst with
  | [] -> acc
  | hd :: tl -> foldleft f (f acc hd) tl ;;
```

Different Data Types

list	<type> list	Group together data of the same type
tuple	(<type1> * <type2>)	Group together data of different types
Record	{field one: <type1>; field two: <type2>;...}	Group together data of different types, but it is easier to access data.
Algebraic Data type	Name_one of <type1> Name_two of <type2>	One piece of data can have many forms



By [benjenkinsv95](#)

Published 9th March, 2017.

Last updated 9th March, 2017.

Page 4 of 5.

Sponsored by [Readable.com](#)

Measure your website readability!

<https://readable.com>

Module

```
module type Integer =  
sig  
val value : int  
end  
  
module ThreeSig : Integer =  
struct  
let value = 3  
let is_positive : bool = true  
end
```

Applying the signature Integer, via ": Integer" restricts ThreeSig makes only "value" public.

Functor

Takes in a module and returns a new module. Ex.

```
module Increment ( I : Integer ) : Integer =  
struct  
let value = I.value + 1  
end
```

Functor can take in multiple modules.

Abstraction Barrier

If a signature has a type

```
module type OPERATORS =  
sig  
type t  
end ;;
```

Then it can be overridden in the implementation using the "with" keyword

```
module IntOps : OPERATORS with type t = int =  
struct  
type t = int  
let times = ( * )  
let divide = ( / )  
let plus = ( + )  
let int_to_t x = x  
end ;;
```



By [benjenkinsv95](#)

Published 9th March, 2017.

Last updated 9th March, 2017.

Page 5 of 5.

Sponsored by [Readable.com](#)

Measure your website readability!

<https://readable.com>