

[Previous](#) [Up](#) [Next](#)

Module List

module List: **sig .. end**

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

val length : 'a list -> int

Return the length (number of elements) of the given list.

val compare_lengths : 'a list -> 'b list -> int

Compare the lengths of two lists. `compare_lengths l1 l2` is equivalent to `compare (length l1) (length l2)`, except that the computation stops after iterating on the shortest list.

Since 4.05.0**val** compare_length_with : 'a list -> int -> int

Compare the length of a list to an integer. `compare_length_with l n` is equivalent to `compare (length l) n`, except that the computation stops after at most `n` iterations on the list.

Since 4.05.0**val** cons : 'a -> 'a list -> 'a list`cons x xs` is `x :: xs`**Since** 4.03.0**val** hd : 'a list -> 'aReturn the first element of the given list. Raise `Failure "hd"` if the list is empty.**val** tl : 'a list -> 'a listReturn the given list without its first element. Raise `Failure "tl"` if the list is empty.**val** nth : 'a list -> int -> 'a

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

val nth_opt : 'a list -> int -> 'a option

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Return `None` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

Since 4.05**val** rev : 'a list -> 'a list

List reversal.

val init : int -> (int -> 'a) -> 'a list`List.init len f` is `f 0; f 1; ...; f (len-1)`, evaluated left to right.

Since 4.06.0

Raises `Invalid_argument` if `len < 0`.

val `append` : 'a list -> 'a list -> 'a list

Concatenate two lists. Same as the infix operator `@`. Not tail-recursive (length of the first argument).

val `rev_append` : 'a list -> 'a list -> 'a list

`List.rev_append l1 l2` reverses `l1` and concatenates it to `l2`. This is equivalent to `List.rev l1 @ l2`, but `rev_append` is tail-recursive and more efficient.

val `concat` : 'a list list -> 'a list

Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

val `flatten` : 'a list list -> 'a list

An alias for `concat`.

Iterators

val `iter` : ('a -> unit) -> 'a list -> unit

`List.iter f [a1; ...; an]` applies function `f` in turn to `a1`; ...; `an`. It is equivalent to `begin f a1; f a2; ...; f an; () end`.

val `iteri` : (int -> 'a -> unit) -> 'a list -> unit

Same as `List.iter`, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.
Since 4.00.0

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

`List.map f [a1; ...; an]` applies function `f` to `a1`, ..., `an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

val `mapi` : (int -> 'a -> 'b) -> 'a list -> 'b list

Same as `List.map`, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument. Not tail-recursive.
Since 4.00.0

val `rev_map` : ('a -> 'b) -> 'a list -> 'b list

`List.rev_map f l` gives the same result as `List.rev (List.map f l)`, but is tail-recursive and more efficient.

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

`List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

val `fold_right` : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

`List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.

Iterators on two lists

val `iter2` : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit

`List.iter2 f [a1; ...; an] [b1; ...; bn]` calls in turn `f a1 b1`; ...; `f an bn`. Raise `Invalid_argument` if the two lists are determined to have different lengths.

val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
 List.map2 f [a1; ...; an] [b1; ...; bn] is [f a1 b1; ...; f an bn]. Raise `Invalid_argument` if the two lists are determined to have different lengths. Not tail-recursive.

val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
 List.rev_map2 f l1 l2 gives the same result as `List.rev (List.map2 f l1 l2)`, but is tail-recursive and more efficient.

val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
 List.fold_left2 f a [b1; ...; bn] [c1; ...; cn] is
 f (... (f (f a b1 c1) b2 c2) ...) bn cn. Raise `Invalid_argument` if the two lists are determined to have different lengths.

val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
 List.fold_right2 f [a1; ...; an] [b1; ...; bn] c is
 f a1 b1 (f a2 b2 (... (f an bn c) ...)). Raise `Invalid_argument` if the two lists are determined to have different lengths. Not tail-recursive.

List scanning

val for_all : ('a -> bool) -> 'a list -> bool
 for_all p [a1; ...; an] checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).

val exists : ('a -> bool) -> 'a list -> bool
 exists p [a1; ...; an] checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).

val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
 Same as `List.for_all`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists are determined to have different lengths.

val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
 Same as `List.exists`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists are determined to have different lengths.

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 instead of structural equality to compare list elements.

List searching

val find : ('a -> bool) -> 'a list -> 'a
 find p l returns the first element of the list l that satisfies the predicate p. Raise `Not_found` if there is no value that satisfies p in the list l.

val find_opt : ('a -> bool) -> 'a list -> 'a option
 find_opt p l returns the first element of the list l that satisfies the predicate p, or `None` if there is no value that satisfies p in the list l.
 Since 4.05

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

`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.

val `find_all` : ('a -> bool) -> 'a list -> 'a list
`find_all` is another name for `List.filter`.

val `partition` : ('a -> bool) -> 'a list -> 'a list * 'a list
`partition p l` 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`. The order of the elements in the input list is preserved.

Association lists

val `assoc` : 'a -> ('a * 'b) list -> 'b
`assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with `a` in the list `l`.

val `assoc_opt` : 'a -> ('a * 'b) list -> 'b option
`assoc_opt a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc_opt a [...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Returns `None` if there is no value associated with `a` in the list `l`.
Since 4.05

val `assq` : 'a -> ('a * 'b) list -> 'b
 Same as `List.assoc`, but uses physical equality instead of structural equality to compare keys.

val `assq_opt` : 'a -> ('a * 'b) list -> 'b option
 Same as `List.assoc_opt`, but uses physical equality instead of structural equality to compare keys.
Since 4.05

val `mem_assoc` : 'a -> ('a * 'b) list -> bool
 Same as `List.assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.

val `mem_assq` : 'a -> ('a * 'b) list -> bool
 Same as `List.mem_assoc`, but uses physical equality instead of structural equality to compare keys.

val `remove_assoc` : 'a -> ('a * 'b) list -> ('a * 'b) list
`remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any. Not tail-recursive.

val `remove_assq` : 'a -> ('a * 'b) list -> ('a * 'b) list
 Same as `List.remove_assoc`, but uses physical equality instead of structural equality to compare keys. Not tail-recursive.

Lists of pairs

val `split` : ('a * 'b) list -> 'a list * 'b list
 Transform a list of pairs into a pair of lists: `split [(a1,b1); ...; (an,bn)]` is `([a1; ...; an], [b1; ...; bn])`. Not tail-recursive.

val `combine` : 'a list -> 'b list -> ('a * 'b) list

Transform a pair of lists into a list of pairs:

`combine [a1; ...; an] [b1; ...; bn]` is `[(a1,b1); ...; (an,bn)]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

Sorting

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

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see `Array.sort` for a complete specification). For example, `compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

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

Same as `List.sort`, but the sorting algorithm is guaranteed to be stable (i.e. elements that compare equal are kept in their original order) .

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

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

Same as `List.sort` or `List.stable_sort`, whichever is faster on typical input.

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

Same as `List.sort`, but also remove duplicates.
Since 4.02.0

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`. If several elements compare equal, the elements of `l1` will be before the elements of `l2`. Not tail-recursive (sum of the lengths of the arguments).

Iterators

val `to_seq` : 'a list -> 'a Seq.t

Iterate on the list
Since 4.07

val `of_seq` : 'a Seq.t -> 'a list

Create a list from the iterator
Since 4.07