OCaml

API Version 5.1

< General Index

OCaml library : List

Comparison

Iterators

Iterators on two lists

List scanning

List searching

Association lists

Lists of pairs

Sorting

Lists and Sequences

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.

The labeled version of this module can be used as described in the `StdLabels` module.

```
type 'a t = 'a list =
| []
| (::) of 'a * 'a list
```

An alias for the type of lists.

```
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 reaching the end of the shortest list.

Since 4.05

```
val compare_length_with : 'a list -> int ->
int
```

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

Since 4.05

```
val is_empty : 'a list -> bool
```

`is_empty l` is true if and only if `l` has no elements.
It is equivalent to `compare_length_with l 0 = 0`.

Since 5.1

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

`cons x xs` is `x :: xs`

Since 4.03 (4.05 in ListLabels)

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

Return the first element of the given list.

Raises `Failure` if the list is empty.

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

Return the given list without its first element.

Raises `Failure` 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.

Raises

- `Failure` if the list is too short.
- `Invalid_argument` 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.

Since 4.05
Raises `Invalid_argument` if `n` is negative.

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

List reversal.

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

`init len f` is `[f 0; f 1; ...; f (len-1)]`, evaluated left to right.

Since 4.06
Raises `Invalid_argument` if `len < 0`.

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

`append l0 l1` appends `l1` to `l0`. Same function as the infix operator `@`.

Since 5.1 this function is tail-recursive.

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

`rev_append l1 l2` reverses `l1` and concatenates it with `l2`. This is equivalent to

`( List.rev l1) @ l2`.

```
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
```

Same as `List.concat`. Not tail-recursive (length of the argument + length of the longest sub-list).

Comparison

```
val equal : ('a -> 'a -> bool) -> 'a list ->
'a list -> bool
```

`equal eq [a1; ...; an] [b1; ..; bm]` holds when the two input lists have the same length, and for each pair of elements `ai`, `bi` at the same position we have `eq ai bi`.

Note: the `eq` function may be called even if the lists have different length. If you know your equality function is costly, you may want to check `List.compare_lengths` first.

Since 4.12

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

`compare cmp [a1; ...; an] [b1; ...; bm]` performs a lexicographic comparison of the two input lists, using the same `'a -> 'a -> int` interface as `compare`:

- `a1 :: l1` is smaller than `a2 :: l2` (negative result) if `a1` is smaller than `a2`, or if they are equal (0 result) and `l1` is smaller than `l2`
- the empty list `[]` is strictly smaller than non-

empty lists

Note: the `cmp` function will be called even if the lists have different lengths.

Since 4.12

Iterators

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

`iter f [a1; ...; an]` applies function `f` in turn to `[a1; ...; an]`. It is equivalent to `f a1; f a2; ...; f an`.

```
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

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

```
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.

Since 4.00

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

`rev_map f l` gives the same result as

`List.rev (List.map f l)`, but is more efficient.

```
val filter_map : ('a -> 'b option) -> 'a list
-> 'b list
```

`filter_map f l` applies `f` to every element of `l`, filters out the `None` elements and returns the list of the arguments of the `Some` elements.

Since 4.08

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

`concat_map f l` gives the same result as `List.concat (List.map f l)`. Tail-recursive.

Since 4.10

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

`fold_left_map` is a combination of `fold_left` and `map` that threads an accumulator through calls to `f`.

Since 4.11

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

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

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

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

Iterators on two lists

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

```
'b list -> unit
```

`iter2 f [a1; ...; an] [b1; ...; bn]` calls in turn `f a1 b1; ...; f an bn`.

Raises `Invalid_argument` if the two lists are determined to have different lengths.

```
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b
list -> 'c list
```

`map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`.

Raises `Invalid_argument` if the two lists are determined to have different lengths.

```
val rev_map2 : ('a -> 'b -> 'c) -> 'a list ->
'b list -> 'c list
```

`rev_map2 f l1 l2` gives the same result as `List.rev ( List.map2 f l1 l2)`, but is more efficient.

```
val fold_left2 : ('acc -> 'a -> 'b -> 'acc) -
> 'acc -> 'a list -> 'b list -> 'acc
```

`fold_left2 f init [a1; ...; an] [b1; ...; bn]` is `f (... (f (f init a1 b1) a2 b2) ...) an bn`.

Raises `Invalid_argument` if the two lists are determined to have different lengths.

```
val fold_right2 : ('a -> 'b -> 'acc -> 'acc)
-> 'a list -> 'b list -> 'acc -> 'acc
```

`fold_right2 f [a1; ...; an] [b1; ...; bn] init` is `f a1 b1 (f a2 b2 (... (f an bn init) ...))`.

Raises `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 f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.

```
val exists : ('a -> bool) -> 'a list -> bool
```

`exists f [a1; ...; an]` checks if at least one element of the list satisfies the predicate `f`. That is, it returns `(f a1) || (f a2) || ... || (f an)` for a non-empty list and `false` if the list is empty.

```
val for_all2 : ('a -> 'b -> bool) -> 'a list
-> 'b list -> bool
```

Same as `List.for_all`, but for a two-argument predicate.

Raises `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.

Raises `Invalid_argument` if the two lists are determined to have different lengths.

```
val mem : 'a -> 'a list -> bool
```

`mem a set` is true if and only if `a` is equal to an element of `set`.

```
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 f l` returns the first element of the list `l` that satisfies the predicate `f`.

Raises `Not_found` if there is no value that satisfies `f` in the list `l`.

```
val find_opt : ('a -> bool) -> 'a list -> 'a
option
```

`find f l` returns the first element of the list `l` that satisfies the predicate `f`. Returns `None` if there is no value that satisfies `f` in the list `l`.

Since 4.05

```
val find_index : ('a -> bool) -> 'a list ->
int option
```

`find_index f xs` returns `Some i`, where `i` is the index of the first element of the list `xs` that satisfies `f x`, if there is such an element.

It returns `None` if there is no such element.

Since 5.1

```
val find_map : ('a -> 'b option) -> 'a list -
> 'b option
```

`find_map f l` applies `f` to the elements of `l` in order, and returns the first result of the form `Some v`, or `None` if none exist.

Since 4.10

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

Same as `find_map`, but the predicate is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.

Since 5.1

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

`filter f l` returns all the elements of the list `l` that satisfy the predicate `f`. 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 filteri : (int -> 'a -> bool) -> 'a list -> 'a list
```

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

Since 4.11

```
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

`partition f l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the elements of `l` that satisfy the predicate `f`, and `l2` is the list of all the elements of `l` that do not satisfy `f`. The order of the elements in the input list is preserved.

```
val partition_map : ('a -> ('b, 'c) Either.t) -> 'a list -> 'b list * 'c list
```

`partition_map f l` returns a pair of lists `(l1, l2)` such that, for each element `x` of the input list `l`:

- if `f x` is `Left y1`, then `y1` is in `l1`, and
- if `f x` is `Right y2`, then `y2` is in `l2`.

The output elements are included in `l1` and `l2` in the same relative order as the corresponding input elements in `l`.

In particular, `partition_map (fun x -> if f x then Left x else Right x) l` is equivalent to `partition f l`.

Since 4.12

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`.

Raises `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); ...] = Some 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)]`.

Raises `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 (4.03 in ListLabels)

```
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).

Lists and Sequences

```
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 a sequence.

Since 4.07