19.16 Module `List`: 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 hd : 'a list -> 'a
```

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

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

Return 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.

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

List reversal.

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

Catenate two lists. Same function as the infix operator `@`. Not tail-recursive (length of the first argument). The `@` operator is not tail-recursive either.

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

`List.rev_append l1 l2` reverses `l1` and catenates 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
val flatten : 'a list list -> 'a list
```

Catenate (flatten) a list of lists. Not tail-recursive (length of the argument + length of the longest sub-list).

Iterators

```
val iter : f:('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 map : f:('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 rev_map : f:('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 : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a
```

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

```
val fold_right : f:('a -> 'b -> 'b) -> 'a list -> init:'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 : f:('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 have different lengths.

```
val map2 : f:('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 have different lengths. Not tail-recursive.

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

`List.rev_map2 f l` gives the same result as `List.rev (List.map2 f l)`, but is tail-recursive and more efficient.

```
val fold_left2 :
        f:('a -> 'b -> 'c -> 'a) -> init:'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 have different lengths.

```
val fold_right2 :
        f:('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> init:'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 have different lengths. Not tail-recursive.

List scanning

```
val for_all : f:('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 : f:('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 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val exists2 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
```

Same as `for_all` and `exists`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists 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 `mem`, but uses physical equality instead of structural equality to compare list elements.

List searching

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

```
val partition : f:('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 assq : 'a -> ('a * 'b) list -> 'b
```

Same as `assoc`, but uses physical equality instead of structural equality to compare keys.

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

Same as `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 `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 `remove_assq`, 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.