Module Array

`module Array: sig .. end`

Array operations.

`val length : 'a array -> int`

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

`val get : 'a array -> int -> 'a`

`Array.get a n` returns the element number `n` of array `a`. The first element has number 0. The last element has number `Array.length a - 1`. You can also write `a.(n)` instead of `Array.get a n`.

Raise `Invalid_argument "index out of bounds"` if `n` is outside the range 0 to `(Array.length a - 1)`.

`val set : 'a array -> int -> 'a -> unit`

`Array.set a n x` modifies array `a` in place, replacing element number `n` with `x`. You can also write `a.(n) <- x` instead of `Array.set a n x`.

Raise `Invalid_argument "index out of bounds"` if `n` is outside the range 0 to `Array.length a - 1`.

`val make : int -> 'a -> 'a array`

`Array.make n x` returns a fresh array of length `n`, initialized with `x`. All the elements of this new array are initially physically equal to `x` (in the sense of the `==` predicate). Consequently, if `x` is mutable, it is shared among all elements of the array, and modifying `x` through one of the array entries will modify all other entries at the same time.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_array_length`. If the value of `x` is a floating-point number, then the maximum size is only `Sys.max_array_length / 2`.

`val create : int -> 'a -> 'a array`

Deprecated.`Array.create` is an alias for `Array.make`.

`val create_float : int -> float array`

`Array.create_float n` returns a fresh float array of length `n`, with uninitialized data.
Since 4.03

`val make_float : int -> float array`

Deprecated.`Array.make_float` is an alias for `Array.create_float`.

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

`Array.init n f` returns a fresh array of length `n`, with element number `i` initialized to the result of `f i`. In other terms, `Array.init n f` tabulates the results of `f` applied to the integers `0` to `n-1`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_array_length`. If the return type of `f` is `float`, then the maximum size is only `Sys.max_array_length / 2`.

`val make_matrix : int -> int -> 'a -> 'a array array`

`Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension `dimx` and second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`. The element (`x,y`) of a matrix `m` is accessed with the notation `m.(x).(y)`.

Raise `Invalid_argument` if `dimx` or `dimy` is negative or greater than `Sys.max_array_length`. If the value of `e` is a floating-point number, then the maximum size is only `Sys.max_array_length / 2`.

`val create_matrix : int -> int -> 'a -> 'a array array`

Deprecated.`Array.create_matrix` is an alias for `Array.make_matrix`.

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

`Array.append v1 v2` returns a fresh array containing the concatenation of the arrays `v1` and `v2`.

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

Same as `Array.append`, but concatenates a list of arrays.

`val sub : 'a array -> int -> int -> 'a array`

`Array.sub a start len` returns a fresh array of length `len`, containing the elements number `start` to `start + len - 1` of array `a`.

Raise `Invalid_argument "Array.sub"` if `start` and `len` do not designate a valid subarray of `a`; that is, if `start < 0`, or `len < 0`, or `start + len > Array.length a`.

`val copy : 'a array -> 'a array`

`Array.copy a` returns a copy of `a`, that is, a fresh array containing the same elements as `a`.

`val fill : 'a array -> int -> int -> 'a -> unit`

`Array.fill a ofs len x` modifies the array `a` in place, storing `x` in elements number `ofs` to `ofs + len - 1`.

Raise `Invalid_argument "Array.fill"` if `ofs` and `len` do not designate a valid subarray of `a`.

`val blit : 'a array -> int -> 'a array -> int -> int -> unit`

`Array.blit v1 o1 v2 o2 len` copies `len` elements from array `v1`, starting at element number `o1`, to array `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same array, and the source and destination chunks overlap.

Raise `Invalid_argument "Array.blit"` if `o1` and `len` do not designate a valid subarray of `v1`, or if `o2` and `len` do not designate a valid subarray of `v2`.

`val to_list : 'a array -> 'a list`

`Array.to_list a` returns the list of all the elements of `a`.

`val of_list : 'a list -> 'a array`

`Array.of_list l` returns a fresh array containing the elements of `l`.

Iterators

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

`Array.iter f a` applies function `f` in turn to all the elements of `a`. It is equivalent to `f a.(0); f a.(1); ...; f a.(Array.length a - 1); ()`.

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

Same as `Array.iter`, but the function is applied with the index of the element as first argument, and the element itself as second argument.

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

`Array.map f a` applies function `f` to all the elements of `a`, and builds an array with the results returned by `f`: `[| f a.(0); f a.(1); ...; f a.(Array.length a - 1) |]`.

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

Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

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

`Array.fold_left f x a` computes `f (... (f (f x a.(0)) a.(1)) ...) a.(n-1)`, where `n` is the length of the array `a`.

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

`Array.fold_right f a x` computes `f a.(0) (f a.(1) ( ... (f a.(n-1) x) ...))`, where `n` is the length of the array `a`.

Iterators on two arrays

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

`Array.iter2 f a b` applies function `f` to all the elements of `a` and `b`. Raise `Invalid_argument` if the arrays are not the same size.
Since 4.03.0

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

`Array.map2 f a b` applies function `f` to all the elements of `a` and `b`, and builds an array with the results returned by `f`: `[| f a.(0) b.(0); ...; f a.(Array.length a - 1) b.(Array.length b - 1)|]`. Raise `Invalid_argument` if the arrays are not the same size.
Since 4.03.0

Array scanning

`val for_all : ('a -> bool) -> 'a array -> bool`

`Array.for_all p [|a1; ...; an|]` checks if all elements of the array satisfy the predicate `p`. That is, it returns `(p a1) && (p a2) && ... && (p an)`.
Since 4.03.0

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

`Array.exists p [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`.
Since 4.03.0

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

`mem a l` is true if and only if `a` is equal to an element of `l`.
Since 4.03.0

`val memq : 'a -> 'a array -> bool`

Same as `Array.mem`, but uses physical equality instead of structural equality to compare array elements.
Since 4.03.0

Sorting

```
val sort : ('a -> 'a -> int) -> 'a array -> unit
```

Sort an array 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 below for a complete specification). For example, `compare` is a suitable comparison function, provided there are no floating-point NaN values in the data. After calling `Array.sort`, the array is sorted in place in increasing order. `Array.sort` is guaranteed to run in constant heap space and (at most) logarithmic stack space.

The current implementation uses Heap Sort. It runs in constant stack space.

Specification of the comparison function: Let `a` be the array and `cmp` the comparison function. The following must be true for all x, y, z in a :

- `cmp x y` > 0 if and only if `cmp y x` < 0
- if `cmp x y` >= 0 and `cmp y z` >= 0 then `cmp x z` >= 0

When `Array.sort` returns, `a` contains the same elements as before, reordered in such a way that for all i and j valid indices of `a` :

- `cmp a.(i) a.(j)` >= 0 if and only if i >= j

```
val stable_sort : ('a -> 'a -> int) -> 'a array -> unit
```

Same as `Array.sort`, but the sorting algorithm is stable (i.e. elements that compare equal are kept in their original order) and not guaranteed to run in constant heap space.

The current implementation uses Merge Sort. It uses `n/2` words of heap space, where `n` is the length of the array. It is usually faster than the current implementation of `Array.sort`.

```
val fast_sort : ('a -> 'a -> int) -> 'a array -> unit
```

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

Iterators

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

Iterate on the array, in increasing order. Modifications of the array during iteration will be reflected in the iterator.
Since 4.07

```
val to_seqi : 'a array -> (int * 'a) Seq.t
```

Iterate on the array, in increasing order, yielding indices along elements. Modifications of the array during iteration will be reflected in the iterator.
Since 4.07

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

Create an array from the generator
Since 4.07