

OCaml library

August 8, 2015

Contents

1	Module CamlinternalMod : Run-time support for recursive modules.	1
2	Module StdLabels : Standard labeled libraries.	1
3	Module Map : Association tables over ordered types.	2
4	Module Scanf : Formatted input functions.	5
5	Module Int32 : 32-bit integers.	36
6	Module Int64 : 64-bit integers.	39
7	Module Array : Array operations.	42
8	Module Stack : Last-in first-out stacks.	45
9	Module ArrayLabels : Array operations.	46
10	Module Digest : MD5 message digest.	49
11	Module Complex : Complex numbers.	51
12	Module Bytes : Byte sequence operations.	52
13	Module Parsing : The run-time library for parsers generated by ocaml yacc.	58
14	Module BytesLabels : Byte sequence operations.	60
15	Module Arg : Parsing of command line arguments.	63
16	Module List : List operations.	66
17	Module Pervasives : The initially opened module.	71
18	Module Queue : First-in first-out queues.	91
19	Module Genlex : A generic lexical analyzer.	92

20 Module CamlinternalFormat	93
21 Module Marshal : Marshaling of data structures.	95
22 Module Sys : System interface.	98
23 Module StringLabels : String operations.	102
24 Module Printf : Formatted output functions.	105
25 Module CamlinternalLazy : Run-time support for lazy values.	108
26 Module ListLabels : List operations.	109
27 Module Hashtbl : Hash tables and hash functions.	113
28 Module MoreLabels : Extra labeled libraries.	120
29 Module Oo : Operations on objects	124
30 Module Gc : Memory management control and statistics; finalised values.	125
31 Module Callback : Registering OCaml values with the C runtime.	130
32 Module Buffer : Extensible buffers.	130
33 Module Sort : Sorting and merging lists.	132
34 Module Lazy : Deferred computations.	133
35 Module Camlinternal00 : Run-time support for objects and classes.	134
36 Module Lexing : The run-time library for lexers generated by ocamllex.	137
37 Module Nativeint : Processor-native integers.	139
38 Module Set : Sets over ordered types.	142
39 Module Format : Pretty printing.	146
40 Module CamlinternalFormatBasics	158
41 Module Obj : Operations on internal representations of values.	164
42 Module Char : Character operations.	165
43 Module Stream : Streams and parsers.	166
44 Module String : String operations.	168

45 Module Filename : Operations on file names.	171
46 Module Printexc : Facilities for printing exceptions and inspecting current call stack.	173
47 Module Random : Pseudo-random number generators (PRNG).	178
48 Module Weak : Arrays of weak pointers and hash tables of weak pointers.	180
49 Module Unix : Interface to the Unix system.	183
50 Module Str : Regular expressions and high-level string processing	215
51 Module Bigarray : Large, multi-dimensional, numerical arrays.	221
52 Module Num : Operation on arbitrary-precision numbers.	237

1 Module CamlinternalMod : Run-time support for recursive modules.

All functions in this module are for system use only, not for the casual user.

```

type shape =
  | Function
  | Lazy
  | Class
  | Module of shape array
  | Value of Obj.t
val init_mod : string * int * int -> shape -> Obj.t
val update_mod : shape -> Obj.t -> Obj.t -> unit

```

2 Module StdLabels : Standard labeled libraries.

This meta-module provides labeled version of the `Array`[7], `Bytes`[12], `List`[16] and `String`[44] modules.

They only differ by their labels. Detailed interfaces can be found in `arrayLabels.mli`, `bytesLabels.mli`, `listLabels.mli` and `stringLabels.mli`.

```

module Array :
  ArrayLabels
module Bytes :
  BytesLabels
module List :
  ListLabels

```

```
module String :
  StringLabels
```

3 Module Map : Association tables over ordered types.

This module implements applicative association tables, also known as finite maps or dictionaries, given a total ordering function over the keys. All operations over maps are purely applicative (no side-effects). The implementation uses balanced binary trees, and therefore searching and insertion take time logarithmic in the size of the map.

For instance:

```
module IntPairs =
  struct
    type t = int * int
    let compare (x0,y0) (x1,y1) =
      match Pervasives.compare x0 x1 with
      | 0 -> Pervasives.compare y0 y1
      | c -> c
  end

module PairsMap = Map.Make(IntPairs)

let m = PairsMap.(empty |> add (0,1) "hello" |> add (1,0) "world")
```

This creates a new module `PairsMap`, with a new type `'a PairsMap.t` of maps from `int * int` to `'a`. In this example, `m` contains `string` values so its type is `string PairsMap.t`.

```
module type OrderedType =
  sig
```

```
    type t
```

The type of the map keys.

```
  val compare : t -> t -> int
```

A total ordering function over the keys. This is a two-argument function `f` such that `f e1 e2` is zero if the keys `e1` and `e2` are equal, `f e1 e2` is strictly negative if `e1` is smaller than `e2`, and `f e1 e2` is strictly positive if `e1` is greater than `e2`. Example: a suitable ordering function is the generic structural comparison function `Pervasives.compare`[17].

```
end
```

Input signature of the functor `Map.Make`[3].

```
module type S =
  sig
```

type `key`

The type of the map keys.

type `+'a t`

The type of maps from type `key` to type `'a`.

val `empty` : `'a t`

The empty map.

val `is_empty` : `'a t -> bool`

Test whether a map is empty or not.

val `mem` : `key -> 'a t -> bool`

`mem x m` returns `true` if `m` contains a binding for `x`, and `false` otherwise.

val `add` : `key -> 'a -> 'a t -> 'a t`

`add x y m` returns a map containing the same bindings as `m`, plus a binding of `x` to `y`. If `x` was already bound in `m`, its previous binding disappears.

val `singleton` : `key -> 'a -> 'a t`

`singleton x y` returns the one-element map that contains a binding `y` for `x`.

Since: 3.12.0

val `remove` : `key -> 'a t -> 'a t`

`remove x m` returns a map containing the same bindings as `m`, except for `x` which is unbound in the returned map.

val `merge` :

`(key -> 'a option -> 'b option -> 'c option) ->
'a t -> 'b t -> 'c t`

`merge f m1 m2` computes a map whose keys is a subset of keys of `m1` and of `m2`. The presence of each such binding, and the corresponding value, is determined with the function `f`.

Since: 3.12.0

val `compare` : `('a -> 'a -> int) -> 'a t -> 'a t -> int`

Total ordering between maps. The first argument is a total ordering used to compare data associated with equal keys in the two maps.

val `equal` : `('a -> 'a -> bool) -> 'a t -> 'a t -> bool`

`equal cmp m1 m2` tests whether the maps `m1` and `m2` are equal, that is, contain equal keys and associate them with equal data. `cmp` is the equality predicate used to compare the data associated with the keys.

`val iter : (key -> 'a -> unit) -> 'a t -> unit`

`iter f m` applies `f` to all bindings in map `m`. `f` receives the key as first argument, and the associated value as second argument. The bindings are passed to `f` in increasing order with respect to the ordering over the type of the keys.

`val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`fold f m a` computes `(f kN dN ... (f k1 d1 a) ...)`, where `k1 ... kN` are the keys of all bindings in `m` (in increasing order), and `d1 ... dN` are the associated data.

`val for_all : (key -> 'a -> bool) -> 'a t -> bool`

`for_all p m` checks if all the bindings of the map satisfy the predicate `p`.

Since: 3.12.0

`val exists : (key -> 'a -> bool) -> 'a t -> bool`

`exists p m` checks if at least one binding of the map satisfy the predicate `p`.

Since: 3.12.0

`val filter : (key -> 'a -> bool) -> 'a t -> 'a t`

`filter p m` returns the map with all the bindings in `m` that satisfy predicate `p`.

Since: 3.12.0

`val partition : (key -> 'a -> bool) -> 'a t -> 'a t * 'a t`

`partition p m` returns a pair of maps `(m1, m2)`, where `m1` contains all the bindings of `s` that satisfy the predicate `p`, and `m2` is the map with all the bindings of `s` that do not satisfy `p`.

Since: 3.12.0

`val cardinal : 'a t -> int`

Return the number of bindings of a map.

Since: 3.12.0

`val bindings : 'a t -> (key * 'a) list`

Return the list of all bindings of the given map. The returned list is sorted in increasing order with respect to the ordering `Ord.compare`, where `Ord` is the argument given to `Map.Make[3]`.

Since: 3.12.0

`val min_binding : 'a t -> key * 'a`

Return the smallest binding of the given map (with respect to the `Ord.compare` ordering), or raise `Not_found` if the map is empty.

Since: 3.12.0

```
val max_binding : 'a t -> key * 'a
```

Same as `Map.S.min_binding[3]`, but returns the largest binding of the given map.

Since: 3.12.0

```
val choose : 'a t -> key * 'a
```

Return one binding of the given map, or raise `Not_found` if the map is empty. Which binding is chosen is unspecified, but equal bindings will be chosen for equal maps.

Since: 3.12.0

```
val split : key -> 'a t -> 'a t * 'a option * 'a t
```

`split x m` returns a triple `(l, data, r)`, where `l` is the map with all the bindings of `m` whose key is strictly less than `x`; `r` is the map with all the bindings of `m` whose key is strictly greater than `x`; `data` is `None` if `m` contains no binding for `x`, or `Some v` if `m` binds `v` to `x`.

Since: 3.12.0

```
val find : key -> 'a t -> 'a
```

`find x m` returns the current binding of `x` in `m`, or raises `Not_found` if no such binding exists.

```
val map : ('a -> 'b) -> 'a t -> 'b t
```

`map f m` returns a map with same domain as `m`, where the associated value `a` of all bindings of `m` has been replaced by the result of the application of `f` to `a`. The bindings are passed to `f` in increasing order with respect to the ordering over the type of the keys.

```
val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
```

Same as `Map.S.map[3]`, but the function receives as arguments both the key and the associated value for each binding of the map.

end

Output signature of the functor `Map.Make[3]`.

```
module Make :
```

```
  functor (Ord : OrderedType) -> S with type key = Ord.t
```

Functor building an implementation of the map structure given a totally ordered type.

4 Module Scanf : Formatted input functions.

Introduction

Introduction

Functional input with format strings

Introduction

Functional input with format strings

The module **Scanf** provides formatted input functions or *scanners*.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a *formatted input channel* (or *scanning buffer*) and has type `Scanf.Scanning.in_channel[4]`. The more general formatted input function reads from any scanning buffer and is named **bscanf**.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a *receiver function* that is applied to the values read.

Hence, a typical call to the formatted input function `Scanf.bscanf[4]` is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a *formatted input channel* with type `Scanf.Scanning.in_channel[4]`)
- `fmt` is a format string (the same format strings as those used to print material with module `Printf[24]` or `Format[39]`),
- `f` is a function that has as many arguments as the number of values to read in the input.

Introduction

Functional input with format strings

The module **Scanf** provides formatted input functions or *scanners*.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a *formatted input channel* (or *scanning buffer*) and has type `Scanf.Scanning.in_channel[4]`. The more general formatted input function reads from any scanning buffer and is named **bscanf**.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a *receiver function* that is applied to the values read.

Hence, a typical call to the formatted input function `Scanf.bscanf[4]` is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a *formatted input channel* with type `Scanf.Scanning.in_channel[4]`)

- `fmt` is a format string (the same format strings as those used to print material with module `Printf`[24] or `Format`[39]),
- `f` is a function that has as many arguments as the number of values to read in the input.

A simple example

Introduction

Functional input with format strings

The module `Scanf` provides formatted input functions or *scanners*.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a *formatted input channel* (or *scanning buffer*) and has type `Scanf.Scanning.in_channel`[4]. The more general formatted input function reads from any scanning buffer and is named `bscanf`.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a *receiver function* that is applied to the values read.

Hence, a typical call to the formatted input function `Scanf.bscanf`[4] is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a *formatted input channel* with type `Scanf.Scanning.in_channel`[4])
- `fmt` is a format string (the same format strings as those used to print material with module `Printf`[24] or `Format`[39]),
- `f` is a function that has as many arguments as the number of values to read in the input.

A simple example

As suggested above, the expression `bscanf ic "%d" f` reads a decimal integer `n` from the source of characters `ic` and returns `f n`.

For instance,

- if we use `stdin` as the source of characters (`Scanf.Scanning.stdin`[4] is the predefined formatted input channel that reads from standard input),
- if we define the receiver `f` as `let f x = x + 1`,

then `bscanf Scanning.stdin "%d" f` reads an integer `n` from the standard input and returns `f n` (that is `n + 1`). Thus, if we evaluate `bscanf stdin "%d" f`, and then enter 41 at the keyboard, we get 42 as the final result.

Introduction

Functional input with format strings

The module `Scanf` provides formatted input functions or *scanners*.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a *formatted input channel* (or *scanning buffer*) and has type `Scanf.Scanning.in_channel`[4]. The more general formatted input function reads from any scanning buffer and is named `bscanf`.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a *receiver function* that is applied to the values read.

Hence, a typical call to the formatted input function `Scanf.bscanf`[4] is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a *formatted input channel* with type `Scanf.Scanning.in_channel`[4]),
- `fmt` is a format string (the same format strings as those used to print material with module `Printf`[24] or `Format`[39]),
- `f` is a function that has as many arguments as the number of values to read in the input.

A simple example

As suggested above, the expression `bscanf ic "%d" f` reads a decimal integer `n` from the source of characters `ic` and returns `f n`.

For instance,

- if we use `stdin` as the source of characters (`Scanf.Scanning.stdin`[4] is the predefined formatted input channel that reads from standard input),
- if we define the receiver `f` as `let f x = x + 1`,

then `bscanf Scanning.stdin "%d" f` reads an integer `n` from the standard input and returns `f n` (that is `n + 1`). Thus, if we evaluate `bscanf stdin "%d" f`, and then enter 41 at the keyboard, we get 42 as the final result.

Formatted input as a functional feature

Introduction

Functional input with format strings

The module `Scanf` provides formatted input functions or *scanners*.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a *formatted input channel* (or *scanning buffer*) and has type `Scanf.Scanning.in_channel`[4]. The more general formatted input function reads from any scanning buffer and is named `bscanf`.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a *receiver function* that is applied to the values read.

Hence, a typical call to the formatted input function `Scanf.bscanf`[4] is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a *formatted input channel* with type `Scanf.Scanning.in_channel`[4])

- `fmt` is a format string (the same format strings as those used to print material with module `Printf`[24] or `Format`[39]),
- `f` is a function that has as many arguments as the number of values to read in the input.

A simple example

As suggested above, the expression `bscanf ic "%d" f` reads a decimal integer `n` from the source of characters `ic` and returns `f n`.

For instance,

- if we use `stdin` as the source of characters (`Scanf.Scanning.stdin`[4] is the predefined formatted input channel that reads from standard input),
- if we define the receiver `f` as `let f x = x + 1`,

then `bscanf Scanning.stdin "%d" f` reads an integer `n` from the standard input and returns `f n` (that is `n + 1`). Thus, if we evaluate `bscanf stdin "%d" f`, and then enter 41 at the keyboard, we get 42 as the final result.

Formatted input as a functional feature

The OCaml scanning facility is reminiscent of the corresponding C feature. However, it is also largely different, simpler, and yet more powerful: the formatted input functions are higher-order functionals and the parameter passing mechanism is just the regular function application not the variable assignment based mechanism which is typical for formatted input in imperative languages; the OCaml format strings also feature useful additions to easily define complex tokens; as expected within a functional programming language, the formatted input functions also support polymorphism, in particular arbitrary interaction with polymorphic user-defined scanners. Furthermore, the OCaml formatted input facility is fully type-checked at compile time.

Introduction

Functional input with format strings

The module `Scanf` provides formatted input functions or *scanners*.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a *formatted input channel* (or *scanning buffer*) and has type `Scanf.Scanning.in_channel`[4]. The more general formatted input function reads from any scanning buffer and is named `bscanf`.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a *receiver function* that is applied to the values read.

Hence, a typical call to the formatted input function `Scanf.bscanf`[4] is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a *formatted input channel* with type `Scanf.Scanning.in_channel`[4])
- `fmt` is a format string (the same format strings as those used to print material with module `Printf`[24] or `Format`[39]),

- `f` is a function that has as many arguments as the number of values to read in the input.

A simple example

As suggested above, the expression `bscanf ic "%d" f` reads a decimal integer `n` from the source of characters `ic` and returns `f n`.

For instance,

- if we use `stdin` as the source of characters (`Scanf.Scanning.stdin[4]` is the predefined formatted input channel that reads from standard input),
- if we define the receiver `f` as `let f x = x + 1,`

then `bscanf Scanning.stdin "%d" f` reads an integer `n` from the standard input and returns `f n` (that is `n + 1`). Thus, if we evaluate `bscanf stdin "%d" f`, and then enter `41` at the keyboard, we get `42` as the final result.

Formatted input as a functional feature

The OCaml scanning facility is reminiscent of the corresponding C feature. However, it is also largely different, simpler, and yet more powerful: the formatted input functions are higher-order functionals and the parameter passing mechanism is just the regular function application not the variable assignment based mechanism which is typical for formatted input in imperative languages; the OCaml format strings also feature useful additions to easily define complex tokens; as expected within a functional programming language, the formatted input functions also support polymorphism, in particular arbitrary interaction with polymorphic user-defined scanners. Furthermore, the OCaml formatted input facility is fully type-checked at compile time.

Formatted input channel

```
module Scanning :
sig
```

```
  type in_channel
```

The notion of input channel for the `Scanf` module: those channels provide all the machinery necessary to read from a given `Pervasives.in_channel` value. A `Scanf.Scanning.in_channel` value is also called a *formatted input channel* or equivalently a *scanning buffer*. The type `scanbuf` below is an alias for `in_channel`.
Since: 3.12.0

```
  type scanbuf = in_channel
```

The type of scanning buffers. A scanning buffer is the source from which a formatted input function gets characters. The scanning buffer holds the current state of the scan, plus a function to get the next char from the input, and a token buffer to store the string matched so far.

Note: a scanning action may often require to examine one character in advance; when this 'lookahead' character does not belong to the token read, it is stored back in the scanning buffer and becomes the next character yet to be read.

```
  val stdin : in_channel
```

The standard input notion for the `Scanf` module. `Scanning.stdin` is the formatted input channel attached to `Pervasives.stdin`.

Note: in the interactive system, when input is read from `stdin`, the newline character that triggers the evaluation is incorporated in the input; thus, the scanning specifications must properly skip this additional newline character (for instance, simply add a `'\n'` as the last character of the format string).

Since: 3.12.0

```
type file_name = string
```

A convenient alias to designate a file name.

Since: 4.00.0

```
val open_in : file_name -> in_channel
```

`Scanning.open_in fname` returns a formatted input channel for bufferized reading in text mode from file `fname`.

Note: `open_in` returns a formatted input channel that efficiently reads characters in large chunks; in contrast, `from_channel` below returns formatted input channels that must read one character at a time, leading to a much slower scanning rate.

Since: 3.12.0

```
val open_in_bin : file_name -> in_channel
```

`Scanning.open_in_bin fname` returns a formatted input channel for bufferized reading in binary mode from file `fname`.

Since: 3.12.0

```
val close_in : in_channel -> unit
```

Closes the `Pervasives.in_channel` associated with the given `Scanning.in_channel` formatted input channel.

Since: 3.12.0

```
val from_file : file_name -> in_channel
```

An alias for `open_in` above.

```
val from_file_bin : string -> in_channel
```

An alias for `open_in_bin` above.

```
val from_string : string -> in_channel
```

`Scanning.from_string s` returns a formatted input channel which reads from the given string. Reading starts from the first character in the string. The end-of-input condition is set when the end of the string is reached.

```
val from_function : (unit -> char) -> in_channel
```

`Scanning.from_function f` returns a formatted input channel with the given function as its reading method.

When scanning needs one more character, the given function is called.

When the function has no more character to provide, it *must* signal an end-of-input condition by raising the exception `End_of_file`.

```
val from_channel : Pervasives.in_channel -> in_channel
```

`Scanning.from_channel ic` returns a formatted input channel which reads from the regular input channel `ic` argument, starting at the current reading position.

```
val end_of_input : in_channel -> bool
```

`Scanning.end_of_input ic` tests the end-of-input condition of the given formatted input channel.

```
val beginning_of_input : in_channel -> bool
```

`Scanning.beginning_of_input ic` tests the beginning of input condition of the given formatted input channel.

```
val name_of_input : in_channel -> string
```

`Scanning.name_of_input ic` returns the name of the character source for the formatted input channel `ic`.

Since: 3.09.0

```
val stdib : in_channel
```

A deprecated alias for `Scanning.stdin`, the scanning buffer reading from `Pervasives.stdin`.

end

Type of formatted input functions

```
type ('a, 'b, 'c, 'd) scanner = ('a, Scanning.in_channel, 'b, 'c, 'a -> 'd, 'd) Pervasives.formatter
```

The type of formatted input scanners: `('a, 'b, 'c, 'd) scanner` is the type of a formatted input function that reads from some formatted input channel according to some format string; more precisely, if `scan` is some formatted input function, then `scan ic fmt f` applies `f` to the arguments specified by the format string `fmt`, when `scan` has read those arguments from the formatted input channel `ic`.

For instance, the `scanf` function below has type `('a, 'b, 'c, 'd) scanner`, since it is a formatted input function that reads from `Scanning.stdin`: `scanf fmt f` applies `f` to the arguments specified by `fmt`, reading those arguments from `Pervasives.stdin` as expected.

If the format `fmt` has some `%r` indications, the corresponding input functions must be provided before the receiver `f` argument. For instance, if `read_elem` is an input function for

values of type `t`, then `bscanf ic "%r;" read_elem f` reads a value `v` of type `t` followed by a `';` character, and returns `f v`.

Since: 3.10.0

exception `Scan_failure of string`

The exception that formatted input functions raise when the input cannot be read according to the given format.

The general formatted input function

```
val bscanf : Scanning.in_channel -> ('a, 'b, 'c, 'd) scanner
```

`bscanf ic fmt r1 ... rN f` reads arguments for the function `f`, from the formatted input channel `ic`, according to the format string `fmt`, and applies `f` to these values. The result of this call to `f` is returned as the result of the entire `bscanf` call. For instance, if `f` is the function `fun s i -> i + 1`, then `Scanf.sscanf "x= 1" "%s = %i" f` returns 2.

Arguments `r1` to `rN` are user-defined input functions that read the argument corresponding to the `%r` conversions specified in the format string.

Format string description

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),

- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (' ' or ASCII code 32) and the line feed character ('\n' or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (' ' or ASCII code 32) and the line feed character ('\n' or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Conversion specifications in format strings

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (' ' or ASCII code 32) and the line feed character ('\n' or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Conversion specifications in format strings

Conversion specifications consist in the % character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- **d**: reads an optionally signed decimal integer.
- **i**: reads an optionally signed integer (usual input conventions for decimal (0-9+), hexadecimal (0x[0-9a-f]+ and 0X[0-9A-F]+), octal (0o[0-7]+), and binary (0b[0-1]+) notations are understood).
- **u**: reads an unsigned decimal integer.
- **x** or **X**: reads an unsigned hexadecimal integer ([0-9a-fA-F]+).
- **o**: reads an unsigned octal integer ([0-7]+).
- **s**: reads a string argument that spreads as much as possible, until the following bounding condition holds:
 - a whitespace has been found (see [4]),
 - a scanning indication (see scanning [4]) has been encountered,
 - the end-of-input has been reached.

Hence, this conversion always succeeds: it returns an empty string if the bounding condition holds when the scan begins.

- **S**: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise `Invalid_argument`, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of OCaml).

- **f, e, E, g, G**: reads an optionally signed floating-point number in decimal notation, in the style `dddd.ddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of OCaml (hence the decimal point is mandatory if the exponent part is not mentioned).
- **B**: reads a boolean argument (`true` or `false`).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).
- **ld, li, lu, lx, lX, lo**: reads an `int32` argument to the format specified by the second letter for regular integers.
- **nd, ni, nu, nx, nX, no**: reads a `nativeint` argument to the format specified by the second letter for regular integers.
- **Ld, Li, Lu, Lx, LX, Lo**: reads an `int64` argument to the format specified by the second letter for regular integers.
- **[range]**: reads characters that matches one of the characters mentioned in the range of characters **range** (or not mentioned in it, if the range starts with `^`). Reads a **string** that can be empty, if the next input character does not match the range. The set of characters from **c1** to **c2** (inclusively) is denoted by **c1-c2**. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly, `%[0-9a-f]` returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the `^` in case of range negation); hence `[]` matches a `]` character and `[^]` matches any character that is not `]`. Use `%%` and `%@` to include a `%` or a `@` in a range.
- **r**: user-defined reader. Takes the next **ri** formatted input function and applies it to the scanning buffer **ib** to read the next argument. The input function **ri** must therefore have type `Scanning.in_channel -> 'a` and the argument read has type `'a`.
- **{ fmt %}**: reads a format string argument. The format string read must have the same type as the format string specification **fmt**. For instance, `"%{ %i %}"` reads any format string that can read a value of type `int`; hence, if **s** is the string `"fmt: \"number is %u\""`, then `Scanf.sscanf s "fmt: %{%i%}"` succeeds and returns the format string `"number is %u"`.
- **(fmt %)**: scanning sub-format substitution. Reads a format string **rf** in the input, then goes on scanning with **rf** instead of scanning with **fmt**. The format string **rf** must have the same type as the format string specification **fmt** that it replaces. For instance, `"%(%i %)"` reads any format string that can read a value of type `int`. The conversion returns the format string read **rf**, and then a value read using **rf**. Hence, if **s** is the string `"\"%4d\"1234.00"`, then `Scanf.sscanf s "%(%i%)"` (fun **fmt** **i** -> **fmt**, **i**) evaluates to `("%4d", 1234)`. This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag `_` to discard the extraneous argument: conversion `%_(fmt %)` reads a format string **rf** and then behaves the same as format string **rf**. Hence, if **s** is the string `"\"%4d\"1234.00"`, then `Scanf.sscanf s "%_(%i%)"` is simply equivalent to `Scanf.sscanf "1234.00" "%4d"`.

- **l**: returns the number of lines read so far.
- **n**: returns the number of characters read so far.
- **N** or **L**: returns the number of tokens read so far.
- **!**: matches the end of input condition.
- **%**: matches one **%** character in the input.
- **@**: matches one **@** character in the input.
- **,**: does nothing.

Following the **%** character that introduces a conversion, there may be the special flag **_**: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if **f** is the function `fun i -> i + 1`, and **s** is the string `"x = 1"`, then `Scanf.sscanf s "%_s = %i" f` returns `2`.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, **%6d** reads an integer, having at most 6 decimal digits; **%4f** reads a float with at most 4 characters; and **%8[\000-\255]** returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

- as mentioned above, a **%s** conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns `""`.
- in addition to the relevant digits, **'_'** characters may appear inside numbers (this is reminiscent to the usual OCaml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.
- the **scanf** facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module **Str**), stream parsers, **ocamllex**-generated lexers, **ocamlyacc**-generated parsers.

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function **f** (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (**' '** or ASCII code 32) and the line feed character (**'\n'** or ASCII code 10). A space does not

match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Conversion specifications in format strings

Conversion specifications consist in the `%` character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- **d**: reads an optionally signed decimal integer.
- **i**: reads an optionally signed integer (usual input conventions for decimal (`0-9+`), hexadecimal (`0x[0-9a-f]+` and `0X[0-9A-F]+`), octal (`0o[0-7]+`), and binary (`0b[0-1]+`) notations are understood).
- **u**: reads an unsigned decimal integer.
- **x** or **X**: reads an unsigned hexadecimal integer (`[0-9a-fA-F]+`).
- **o**: reads an unsigned octal integer (`[0-7]+`).
- **s**: reads a string argument that spreads as much as possible, until the following bounding condition holds:
 - a whitespace has been found (see [4]),
 - a scanning indication (see scanning [4]) has been encountered,
 - the end-of-input has been reached.

Hence, this conversion always succeeds: it returns an empty string if the bounding condition holds when the scan begins.

- **S**: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise `Invalid_argument`, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **f**, **e**, **E**, **g**, **G**: reads an optionally signed floating-point number in decimal notation, in the style `dddd.ddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of OCaml (hence the decimal point is mandatory if the exponent part is not mentioned).

- **B**: reads a boolean argument (`true` or `false`).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).
- **ld, li, lu, lx, lX, lo**: reads an `int32` argument to the format specified by the second letter for regular integers.
- **nd, ni, nu, nx, nX, no**: reads a `nativeint` argument to the format specified by the second letter for regular integers.
- **Ld, Li, Lu, Lx, LX, Lo**: reads an `int64` argument to the format specified by the second letter for regular integers.
- **[range]**: reads characters that matches one of the characters mentioned in the range of characters **range** (or not mentioned in it, if the range starts with `^`). Reads a **string** that can be empty, if the next input character does not match the range. The set of characters from **c1** to **c2** (inclusively) is denoted by **c1-c2**. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly, `%[0-9a-f]` returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the `^` in case of range negation); hence `[]` matches a `]` character and `[^]` matches any character that is not `]`. Use `%%` and `%@` to include a `%` or a `@` in a range.
- **r**: user-defined reader. Takes the next **ri** formatted input function and applies it to the scanning buffer **ib** to read the next argument. The input function **ri** must therefore have type `Scanning.in_channel -> 'a` and the argument read has type `'a`.
- **{ fmt %}**: reads a format string argument. The format string read must have the same type as the format string specification **fmt**. For instance, `"%{ %i %}"` reads any format string that can read a value of type `int`; hence, if **s** is the string `"fmt:\\"number is %u\\""`, then `Scanf.sscanf s "fmt: %{i%}"` succeeds and returns the format string `"number is %u"`.
- **(fmt %)**: scanning sub-format substitution. Reads a format string **rf** in the input, then goes on scanning with **rf** instead of scanning with **fmt**. The format string **rf** must have the same type as the format string specification **fmt** that it replaces. For instance, `"%(%i %)"` reads any format string that can read a value of type `int`. The conversion returns the format string read **rf**, and then a value read using **rf**. Hence, if **s** is the string `"\\\"%4d\\\"1234.00"`, then `Scanf.sscanf s "%(%i%)"` (fun `fmt i -> fmt, i`) evaluates to `("%4d", 1234)`. This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag `_` to discard the extraneous argument: conversion `%_(fmt %)` reads a format string **rf** and then behaves the same as format string **rf**. Hence, if **s** is the string `"\\\"%4d\\\"1234.00"`, then `Scanf.sscanf s "%_(%i%)"` is simply equivalent to `Scanf.sscanf "1234.00" "%4d"`.
- **l**: returns the number of lines read so far.
- **n**: returns the number of characters read so far.
- **N** or **L**: returns the number of tokens read so far.

- `!`: matches the end of input condition.
- `%`: matches one `%` character in the input.
- `@`: matches one `@` character in the input.
- `,`: does nothing.

Following the `%` character that introduces a conversion, there may be the special flag `_`: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if `f` is the function `fun i -> i + 1`, and `s` is the string `"x = 1"`, then `Scanf.sscanf s "%_s = %i" f` returns 2.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, `%6d` reads an integer, having at most 6 decimal digits; `%4f` reads a float with at most 4 characters; and `%8[\000-\255]` returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

- as mentioned above, a `%s` conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns `""`.
- in addition to the relevant digits, `'_'` characters may appear inside numbers (this is reminiscent to the usual OCaml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.
- the `scanf` facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module `Str`), stream parsers, `ocamllex`-generated lexers, `ocamlyacc`-generated parsers.

Scanning indications in format strings

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (`' '` or ASCII code 32) and the line feed character (`'\n'` or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Conversion specifications in format strings

Conversion specifications consist in the `%` character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- **d**: reads an optionally signed decimal integer.
- **i**: reads an optionally signed integer (usual input conventions for decimal (`0-9+`), hexadecimal (`0x[0-9a-f]+` and `0X[0-9A-F]+`), octal (`0o[0-7]+`), and binary (`0b[0-1]+`) notations are understood).
- **u**: reads an unsigned decimal integer.
- **x** or **X**: reads an unsigned hexadecimal integer (`[0-9a-fA-F]+`).
- **o**: reads an unsigned octal integer (`[0-7]+`).
- **s**: reads a string argument that spreads as much as possible, until the following bounding condition holds:
 - a whitespace has been found (see [4]),
 - a scanning indication (see scanning [4]) has been encountered,
 - the end-of-input has been reached.

Hence, this conversion always succeeds: it returns an empty string if the bounding condition holds when the scan begins.

- **S**: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise `Invalid_argument`, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **f**, **e**, **E**, **g**, **G**: reads an optionally signed floating-point number in decimal notation, in the style `ddd.dddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of OCaml (hence the decimal point is mandatory if the exponent part is not mentioned).
- **B**: reads a boolean argument (`true` or `false`).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).

- **ld, li, lu, lx, lX, lo**: reads an `int32` argument to the format specified by the second letter for regular integers.
- **nd, ni, nu, nx, nX, no**: reads a `nativeint` argument to the format specified by the second letter for regular integers.
- **Ld, Li, Lu, Lx, LX, Lo**: reads an `int64` argument to the format specified by the second letter for regular integers.
- **[range]**: reads characters that matches one of the characters mentioned in the range of characters **range** (or not mentioned in it, if the range starts with `^`). Reads a **string** that can be empty, if the next input character does not match the range. The set of characters from **c1** to **c2** (inclusively) is denoted by **c1-c2**. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly, `%[0-9a-f]` returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the `^` in case of range negation); hence `[]` matches a `]` character and `[^]` matches any character that is not `]`. Use `%%` and `%@` to include a `%` or a `@` in a range.
- **r**: user-defined reader. Takes the next **ri** formatted input function and applies it to the scanning buffer **ib** to read the next argument. The input function **ri** must therefore have type `Scanning.in_channel -> 'a` and the argument read has type `'a`.
- **{ fmt %}**: reads a format string argument. The format string read must have the same type as the format string specification **fmt**. For instance, `"%{ %i %}"` reads any format string that can read a value of type `int`; hence, if **s** is the string `"fmt:\number is %u\"`, then `Scanf.sscanf s "fmt: %{i%}"` succeeds and returns the format string `"number is %u"`.
- **(fmt %)**: scanning sub-format substitution. Reads a format string **rf** in the input, then goes on scanning with **rf** instead of scanning with **fmt**. The format string **rf** must have the same type as the format string specification **fmt** that it replaces. For instance, `"%(%i %)"` reads any format string that can read a value of type `int`. The conversion returns the format string read **rf**, and then a value read using **rf**. Hence, if **s** is the string `"\%4d\1234.00"`, then `Scanf.sscanf s "%(i%)"` (fun `fmt i -> fmt, i`) evaluates to `("%4d", 1234)`. This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag `_` to discard the extraneous argument: conversion `%_(fmt %)` reads a format string **rf** and then behaves the same as format string **rf**. Hence, if **s** is the string `"\%4d\1234.00"`, then `Scanf.sscanf s "%_(i%)"` is simply equivalent to `Scanf.sscanf "1234.00" "%4d"`.
- **l**: returns the number of lines read so far.
- **n**: returns the number of characters read so far.
- **N** or **L**: returns the number of tokens read so far.
- **!**: matches the end of input condition.
- **%**: matches one `%` character in the input.

- `@`: matches one `@` character in the input.
- `,:` does nothing.

Following the `%` character that introduces a conversion, there may be the special flag `_`: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if `f` is the function `fun i -> i + 1`, and `s` is the string `"x = 1"`, then `Scanf.sscanf s "%_s = %i" f` returns 2.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, `%6d` reads an integer, having at most 6 decimal digits; `%4f` reads a float with at most 4 characters; and `%8[\000-\255]` returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

- as mentioned above, a `%s` conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns `""`.
- in addition to the relevant digits, `'_'` characters may appear inside numbers (this is reminiscent to the usual OCaml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.
- the `scanf` facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module `Str`), stream parsers, `ocamllex`-generated lexers, `ocamlyacc`-generated parsers.

Scanning indications in format strings

Scanning indications appear just after the string conversions `%s` and `%[range]` to delimit the end of the token. A scanning indication is introduced by a `@` character, followed by some plain character `c`. It means that the string token should end just before the next matching `c` (which is skipped). If no `c` character is encountered, the string token spreads as much as possible. For instance, `"%s@t"` reads a string up to the next tab character or to the end of input. If a `@` character appears anywhere else in the format string, it is treated as a plain character.

Note:

- As usual in format strings, `%` and `@` characters must be escaped using `%%` and `%@`; this rule still holds within range specifications and scanning indications. For instance, `"%s@%"` reads a string up to the next `%` character.
- The scanning indications introduce slight differences in the syntax of `Scanf` format strings, compared to those used for the `Printf` module. However, the scanning indications are similar to those used in the `Format` module; hence, when producing formatted text to be scanned by `!Scanf.bscanf`, it is wise to use printing functions from the `Format` module (or, if you need to use functions from `Printf`, banish or carefully double check the format strings that contain `'@'` characters).

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (' ' or ASCII code 32) and the line feed character ('\n' or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Conversion specifications in format strings

Conversion specifications consist in the `%` character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- `d`: reads an optionally signed decimal integer.
- `i`: reads an optionally signed integer (usual input conventions for decimal (`0-9+`), hexadecimal (`0x[0-9a-f]+` and `0X[0-9A-F]+`), octal (`0o[0-7]+`), and binary (`0b[0-1]+`) notations are understood).
- `u`: reads an unsigned decimal integer.
- `x` or `X`: reads an unsigned hexadecimal integer (`[0-9a-fA-F]+`).
- `o`: reads an unsigned octal integer (`[0-7]+`).
- `s`: reads a string argument that spreads as much as possible, until the following bounding condition holds:
 - a whitespace has been found (see [4]),
 - a scanning indication (see scanning [4]) has been encountered,
 - the end-of-input has been reached.

Hence, this conversion always succeeds: it returns an empty string if the bounding condition holds when the scan begins.

- `S`: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of OCaml).

- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise `Invalid_argument`, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **f**, **e**, **E**, **g**, **G**: reads an optionally signed floating-point number in decimal notation, in the style `dddd.ddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of OCaml (hence the decimal point is mandatory if the exponent part is not mentioned).
- **B**: reads a boolean argument (`true` or `false`).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).
- **ld**, **li**, **lu**, **lx**, **lX**, **lo**: reads an `int32` argument to the format specified by the second letter for regular integers.
- **nd**, **ni**, **nu**, **nx**, **nX**, **no**: reads a `nativeint` argument to the format specified by the second letter for regular integers.
- **Ld**, **Li**, **Lu**, **Lx**, **LX**, **Lo**: reads an `int64` argument to the format specified by the second letter for regular integers.
- **[range]**: reads characters that matches one of the characters mentioned in the range of characters **range** (or not mentioned in it, if the range starts with `^`). Reads a **string** that can be empty, if the next input character does not match the range. The set of characters from **c1** to **c2** (inclusively) is denoted by **c1-c2**. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly, `%[0-9a-f]` returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the `^` in case of range negation); hence `[]` matches a `]` character and `[^]` matches any character that is not `]`. Use `%%` and `%@` to include a `%` or a `@` in a range.
- **r**: user-defined reader. Takes the next **ri** formatted input function and applies it to the scanning buffer **ib** to read the next argument. The input function **ri** must therefore have type `Scanning.in_channel -> 'a` and the argument read has type `'a`.
- **{ fmt %}**: reads a format string argument. The format string read must have the same type as the format string specification **fmt**. For instance, `"%{ %i %}"` reads any format string that can read a value of type `int`; hence, if **s** is the string `"fmt:\\"number is %u\\""`, then `Scanf.sscanf s "fmt: %{i%}"` succeeds and returns the format string `"number is %u"`.
- **(fmt %)**: scanning sub-format substitution. Reads a format string **rf** in the input, then goes on scanning with **rf** instead of scanning with **fmt**. The format string **rf** must have the same type as the format string specification **fmt** that it replaces. For instance, `"%(%i %)"` reads any format string that can read a value of type `int`. The conversion returns the format string read **rf**, and then a value read using **rf**. Hence, if **s** is the string `"\\\"%4d\\\"1234.00"`,

then `Scanf.sscanf s "%(i%)" (fun fmt i -> fmt, i)` evaluates to `("%4d", 1234)`. This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag `_` to discard the extraneous argument: conversion `%(_ fmt %)` reads a format string `rf` and then behaves the same as format string `rf`. Hence, if `s` is the string `"\"%4d\"1234.00"`, then `Scanf.sscanf s "%_(%i%)"` is simply equivalent to `Scanf.sscanf "1234.00" "%4d"`.

- `l`: returns the number of lines read so far.
- `n`: returns the number of characters read so far.
- `N` or `L`: returns the number of tokens read so far.
- `!`: matches the end of input condition.
- `%`: matches one `%` character in the input.
- `@`: matches one `@` character in the input.
- `,`: does nothing.

Following the `%` character that introduces a conversion, there may be the special flag `_`: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if `f` is the function `fun i -> i + 1`, and `s` is the string `"x = 1"`, then `Scanf.sscanf s "%_s = %i" f` returns `2`.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, `%6d` reads an integer, having at most 6 decimal digits; `%4f` reads a float with at most 4 characters; and `%8[\000-\255]` returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

- as mentioned above, a `%s` conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns `""`.
- in addition to the relevant digits, `'_'` characters may appear inside numbers (this is reminiscent to the usual OCaml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.
- the `scanf` facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module `Str`), stream parsers, `ocamllex`-generated lexers, `ocamlyacc`-generated parsers.

Scanning indications in format strings

Scanning indications appear just after the string conversions `%s` and `%[range]` to delimit the end of the token. A scanning indication is introduced by a `@` character, followed by some plain character `c`. It means that the string token should end just before the next matching `c` (which is skipped). If no `c` character is encountered, the string token spreads as much as possible. For instance, `"%s@t"` reads a string up to the next tab character or to the end of input. If a `@` character appears anywhere else in the format string, it is treated as a plain character.

Note:

- As usual in format strings, % and @ characters must be escaped using %% and %@; this rule still holds within range specifications and scanning indications. For instance, "%s@%" reads a string up to the next % character.
- The scanning indications introduce slight differences in the syntax of **Scanf** format strings, compared to those used for the **Printf** module. However, the scanning indications are similar to those used in the **Format** module; hence, when producing formatted text to be scanned by **!Scanf.bscanf**, it is wise to use printing functions from the **Format** module (or, if you need to use functions from **Printf**, banish or carefully double check the format strings that contain '@' characters).

Exceptions during scanning

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function **f** (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (' ' or ASCII code 32) and the line feed character ('\n' or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call **bscanf ib "Price = %d \$" (fun p -> p)** succeeds and returns 1 when reading an input with various whitespace in it, such as **Price = 1 \$**, **Price = 1 \$**, or even **Price=1\$**.

Conversion specifications in format strings

Conversion specifications consist in the % character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- **d**: reads an optionally signed decimal integer.
- **i**: reads an optionally signed integer (usual input conventions for decimal (0-9+), hexadecimal (0x[0-9a-f]+ and 0X[0-9A-F]+), octal (0o[0-7]+), and binary (0b[0-1]+) notations are understood).
- **u**: reads an unsigned decimal integer.
- **x** or **X**: reads an unsigned hexadecimal integer ([0-9a-fA-F]+).

- **o**: reads an unsigned octal integer (`[0-7]+`).
- **s**: reads a string argument that spreads as much as possible, until the following bounding condition holds:
 - a whitespace has been found (see [4]),
 - a scanning indication (see scanning [4]) has been encountered,
 - the end-of-input has been reached.

Hence, this conversion always succeeds: it returns an empty string if the bounding condition holds when the scan begins.

- **S**: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise `Invalid_argument`, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **f, e, E, g, G**: reads an optionally signed floating-point number in decimal notation, in the style `dddd.ddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of OCaml (hence the decimal point is mandatory if the exponent part is not mentioned).
- **B**: reads a boolean argument (`true` or `false`).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).
- **ld, li, lu, lx, lX, lo**: reads an `int32` argument to the format specified by the second letter for regular integers.
- **nd, ni, nu, nx, nX, no**: reads a `nativeint` argument to the format specified by the second letter for regular integers.
- **Ld, Li, Lu, Lx, LX, Lo**: reads an `int64` argument to the format specified by the second letter for regular integers.
- **[range]**: reads characters that matches one of the characters mentioned in the range of characters `range` (or not mentioned in it, if the range starts with `^`). Reads a `string` that can be empty, if the next input character does not match the range. The set of characters from `c1` to `c2` (inclusively) is denoted by `c1-c2`. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly, `%[0-9a-f]` returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the `^` in case of range negation); hence `[]` matches a `]` character and `[^]` matches any character that is not `]`. Use `%%` and `%@` to include a `%` or a `@` in a range.

- **r**: user-defined reader. Takes the next **ri** formatted input function and applies it to the scanning buffer **ib** to read the next argument. The input function **ri** must therefore have type `Scanning.in_channel -> 'a` and the argument read has type `'a`.
- **{ fmt %}**: reads a format string argument. The format string read must have the same type as the format string specification **fmt**. For instance, **"{%i %}"** reads any format string that can read a value of type `int`; hence, if **s** is the string **"fmt:\number is %u\"**, then `Scanf.sscanf s "fmt: {%i%}"` succeeds and returns the format string **"number is %u"**.
- **(fmt %)**: scanning sub-format substitution. Reads a format string **rf** in the input, then goes on scanning with **rf** instead of scanning with **fmt**. The format string **rf** must have the same type as the format string specification **fmt** that it replaces. For instance, **"(%i %)"** reads any format string that can read a value of type `int`. The conversion returns the format string read **rf**, and then a value read using **rf**. Hence, if **s** is the string **"\%4d\1234.00"**, then `Scanf.sscanf s "%(%i%)"` (fun **fmt** **i** -> **fmt**, **i**) evaluates to **("%4d", 1234)**. This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag **_** to discard the extraneous argument: conversion **%(fmt %)** reads a format string **rf** and then behaves the same as format string **rf**. Hence, if **s** is the string **"\%4d\1234.00"**, then `Scanf.sscanf s "%_(%i%)"` is simply equivalent to `Scanf.sscanf "1234.00" "%4d"`.
- **l**: returns the number of lines read so far.
- **n**: returns the number of characters read so far.
- **N** or **L**: returns the number of tokens read so far.
- **!**: matches the end of input condition.
- **%**: matches one **%** character in the input.
- **@**: matches one **@** character in the input.
- **,:** does nothing.

Following the **%** character that introduces a conversion, there may be the special flag **_**: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if **f** is the function `fun i -> i + 1`, and **s** is the string **"x = 1"**, then `Scanf.sscanf s "%_s = %i" f` returns 2.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, **%6d** reads an integer, having at most 6 decimal digits; **%4f** reads a float with at most 4 characters; and **%8[\000-\255]** returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

- as mentioned above, a **%s** conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns **""**.

- in addition to the relevant digits, `'_'` characters may appear inside numbers (this is reminiscent to the usual OCaml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.
- the `scanf` facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module `Str`), stream parsers, `ocamllex`-generated lexers, `ocamlyacc`-generated parsers.

Scanning indications in format strings

Scanning indications appear just after the string conversions `%s` and `%[range]` to delimit the end of the token. A scanning indication is introduced by a `@` character, followed by some plain character `c`. It means that the string token should end just before the next matching `c` (which is skipped). If no `c` character is encountered, the string token spreads as much as possible. For instance, `"%s@t"` reads a string up to the next tab character or to the end of input. If a `@` character appears anywhere else in the format string, it is treated as a plain character.

Note:

- As usual in format strings, `%` and `@` characters must be escaped using `%%` and `%@`; this rule still holds within range specifications and scanning indications. For instance, `"%s@%"` reads a string up to the next `%` character.
- The scanning indications introduce slight differences in the syntax of `Scanf` format strings, compared to those used for the `Printf` module. However, the scanning indications are similar to those used in the `Format` module; hence, when producing formatted text to be scanned by `!Scanf.bscanf`, it is wise to use printing functions from the `Format` module (or, if you need to use functions from `Printf`, banish or carefully double check the format strings that contain `'@'` characters).

Exceptions during scanning

Scanners may raise the following exceptions when the input cannot be read according to the format string:

- Raise `Scanf.Scan_failure` if the input does not match the format.
- Raise `Failure` if a conversion to a number is not possible.
- Raise `End_of_file` if the end of input is encountered while some more characters are needed to read the current conversion specification.
- Raise `Invalid_argument` if the format string is invalid.

Note:

- as a consequence, scanning a `%s` conversion never raises exception `End_of_file`: if the end of input is reached the conversion succeeds and simply returns the characters read so far, or `""` if none were ever read.

Format string description

The format string is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [4]),
- conversion specifications, each of which causes reading and conversion of one argument for the function `f` (see [4]),
- scanning indications to specify boundaries of tokens (see scanning [4]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (' ' or ASCII code 32) and the line feed character ('\n' or ASCII code 10). A space does not match a single space character, but any amount of 'whitespace' in the input. More precisely, a space inside the format string matches *any number* of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching *any* amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call `bscanf ib "Price = %d $" (fun p -> p)` succeeds and returns 1 when reading an input with various whitespace in it, such as `Price = 1 $`, `Price = 1 $`, or even `Price=1$`.

Conversion specifications in format strings

Conversion specifications consist in the `%` character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- `d`: reads an optionally signed decimal integer.
- `i`: reads an optionally signed integer (usual input conventions for decimal (`0-9+`), hexadecimal (`0x[0-9a-f]+` and `0X[0-9A-F]+`), octal (`0o[0-7]+`), and binary (`0b[0-1]+`) notations are understood).
- `u`: reads an unsigned decimal integer.
- `x` or `X`: reads an unsigned hexadecimal integer (`[0-9a-fA-F]+`).
- `o`: reads an unsigned octal integer (`[0-7]+`).
- `s`: reads a string argument that spreads as much as possible, until the following bounding condition holds:
 - a whitespace has been found (see [4]),
 - a scanning indication (see scanning [4]) has been encountered,
 - the end-of-input has been reached.

Hence, this conversion always succeeds: it returns an empty string if the bounding condition holds when the scan begins.

- `S`: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of OCaml).

- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise `Invalid_argument`, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of OCaml).
- **f**, **e**, **E**, **g**, **G**: reads an optionally signed floating-point number in decimal notation, in the style `dddd.ddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of OCaml (hence the decimal point is mandatory if the exponent part is not mentioned).
- **B**: reads a boolean argument (`true` or `false`).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).
- **ld**, **li**, **lu**, **lx**, **lX**, **lo**: reads an `int32` argument to the format specified by the second letter for regular integers.
- **nd**, **ni**, **nu**, **nx**, **nX**, **no**: reads a `nativeint` argument to the format specified by the second letter for regular integers.
- **Ld**, **Li**, **Lu**, **Lx**, **LX**, **Lo**: reads an `int64` argument to the format specified by the second letter for regular integers.
- **[range]**: reads characters that matches one of the characters mentioned in the range of characters **range** (or not mentioned in it, if the range starts with `^`). Reads a **string** that can be empty, if the next input character does not match the range. The set of characters from **c1** to **c2** (inclusively) is denoted by **c1-c2**. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly, `%[0-9a-f]` returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the `^` in case of range negation); hence `[]` matches a `]` character and `[^]` matches any character that is not `]`. Use `%%` and `%@` to include a `%` or a `@` in a range.
- **r**: user-defined reader. Takes the next **ri** formatted input function and applies it to the scanning buffer **ib** to read the next argument. The input function **ri** must therefore have type `Scanning.in_channel -> 'a` and the argument read has type `'a`.
- **{ fmt %}**: reads a format string argument. The format string read must have the same type as the format string specification **fmt**. For instance, `"%{ %i %}"` reads any format string that can read a value of type `int`; hence, if **s** is the string `"fmt:\\"number is %u\\""`, then `Scanf.sscanf s "fmt: %{i%}"` succeeds and returns the format string `"number is %u"`.
- **(fmt %)**: scanning sub-format substitution. Reads a format string **rf** in the input, then goes on scanning with **rf** instead of scanning with **fmt**. The format string **rf** must have the same type as the format string specification **fmt** that it replaces. For instance, `"%(%i %)"` reads any format string that can read a value of type `int`. The conversion returns the format string read **rf**, and then a value read using **rf**. Hence, if **s** is the string `"\\\"%4d\\\"1234.00"`,

then `Scanf.sscanf s "%(i%)" (fun fmt i -> fmt, i)` evaluates to `("%4d", 1234)`. This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag `_` to discard the extraneous argument: conversion `%_(fmt %)` reads a format string `rf` and then behaves the same as format string `rf`. Hence, if `s` is the string `"\"%4d\"1234.00"`, then `Scanf.sscanf s "%_(i%)"` is simply equivalent to `Scanf.sscanf "1234.00" "%4d"`.

- `l`: returns the number of lines read so far.
- `n`: returns the number of characters read so far.
- `N` or `L`: returns the number of tokens read so far.
- `!`: matches the end of input condition.
- `%`: matches one `%` character in the input.
- `@`: matches one `@` character in the input.
- `,`: does nothing.

Following the `%` character that introduces a conversion, there may be the special flag `_`: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if `f` is the function `fun i -> i + 1`, and `s` is the string `"x = 1"`, then `Scanf.sscanf s "%_s = %i" f` returns `2`.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, `%6d` reads an integer, having at most 6 decimal digits; `%4f` reads a float with at most 4 characters; and `%8[\000-\255]` returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

- as mentioned above, a `%s` conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns `""`.
- in addition to the relevant digits, `'_'` characters may appear inside numbers (this is reminiscent to the usual OCaml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.
- the `scanf` facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module `Str`), stream parsers, `ocamllex`-generated lexers, `ocamlyacc`-generated parsers.

Scanning indications in format strings

Scanning indications appear just after the string conversions `%s` and `%[range]` to delimit the end of the token. A scanning indication is introduced by a `@` character, followed by some plain character `c`. It means that the string token should end just before the next matching `c` (which is skipped). If no `c` character is encountered, the string token spreads as much as possible. For instance, `"%s@t"` reads a string up to the next tab character or to the end of input. If a `@` character appears anywhere else in the format string, it is treated as a plain character.

Note:

- As usual in format strings, % and @ characters must be escaped using %% and %@; this rule still holds within range specifications and scanning indications. For instance, "%s@%" reads a string up to the next % character.
- The scanning indications introduce slight differences in the syntax of `Scanf` format strings, compared to those used for the `Printf` module. However, the scanning indications are similar to those used in the `Format` module; hence, when producing formatted text to be scanned by `!Scanf.bscanf`, it is wise to use printing functions from the `Format` module (or, if you need to use functions from `Printf`, banish or carefully double check the format strings that contain '@' characters).

Exceptions during scanning

Scanners may raise the following exceptions when the input cannot be read according to the format string:

- Raise `Scanf.Scan_failure` if the input does not match the format.
- Raise `Failure` if a conversion to a number is not possible.
- Raise `End_of_file` if the end of input is encountered while some more characters are needed to read the current conversion specification.
- Raise `Invalid_argument` if the format string is invalid.

Note:

- as a consequence, scanning a %s conversion never raises exception `End_of_file`: if the end of input is reached the conversion succeeds and simply returns the characters read so far, or "" if none were ever read.

Specialised formatted input functions

```
val fscanf : Pervasives.in_channel -> ('a, 'b, 'c, 'd) scanner
```

Same as `Scanf.bscanf[4]`, but reads from the given regular input channel.

Warning: since all formatted input functions operate from a *formatted input channel*, be aware that each `fscanf` invocation will operate with a formatted input channel reading from the given channel. This extra level of bufferization can lead to a strange scanning behaviour if you use low level primitives on the channel (reading characters, seeking the reading position, and so on).

As a consequence, never mix direct low level reading and high level scanning from the same regular input channel.

```
val sscanf : string -> ('a, 'b, 'c, 'd) scanner
```

Same as `Scanf.bscanf[4]`, but reads from the given string.

```
val scanf : ('a, 'b, 'c, 'd) scanner
```

Same as `Scanf.bscanf[4]`, but reads from the predefined formatted input channel `Scanf.Scanning.stdin[4]` that is connected to `Pervasives.stdin`.

```
val kscanf :  
  Scanning.in_channel ->  
  (Scanning.in_channel -> exn -> 'd) -> ('a, 'b, 'c, 'd) scanner  
  Same as Scanf.bscanf[4], but takes an additional function argument ef that is called in  
  case of error: if the scanning process or some conversion fails, the scanning function aborts  
  and calls the error handling function ef with the formatted input channel and the exception  
  that aborted the scanning process as arguments.
```

```
val ksscanf :  
  string ->  
  (Scanning.in_channel -> exn -> 'd) -> ('a, 'b, 'c, 'd) scanner  
  Same as Scanf.kscanf[4] but reads from the given string.  
  Since: 4.02.0
```

```
val kfscanf :  
  Pervasives.in_channel ->  
  (Scanning.in_channel -> exn -> 'd) -> ('a, 'b, 'c, 'd) scanner  
  Same as Scanf.kscanf[4], but reads from the given regular input channel.  
  Since: 4.02.0
```

Reading format strings from input

```
val bscanf_format :  
  Scanning.in_channel ->  
  ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 ->  
  (('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 -> 'g) -> 'g  
  bscanf_format ic fmt f reads a format string token from the formatted input channel ic,  
  according to the given format string fmt, and applies f to the resulting format string value.  
  Raise Scan_failure if the format string value read does not have the same type as fmt.  
  Since: 3.09.0
```

```
val sscanf_format :  
  string ->  
  ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 ->  
  (('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 -> 'g) -> 'g  
  Same as Scanf.bscanf_format[4], but reads from the given string.  
  Since: 3.09.0
```

```
val format_from_string :  
  string ->  
  ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 ->  
  ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6
```

`format_from_string s fmt` converts a string argument to a format string, according to the given format string `fmt`. Raise `Scan_failure` if `s`, considered as a format string, does not have the same type as `fmt`.

Since: 3.10.0

`val unescaped : string -> string`

Return a copy of the argument with escape sequences, following the lexical conventions of OCaml, replaced by their corresponding special characters. If there is no escape sequence in the argument, still return a copy, contrary to `String.escaped`.

Since: 4.00.0

5 Module Int32 : 32-bit integers.

This module provides operations on the type `int32` of signed 32-bit integers. Unlike the built-in `int` type, the type `int32` is guaranteed to be exactly 32-bit wide on all platforms. All arithmetic operations over `int32` are taken modulo 2^{32} .

Performance notice: values of type `int32` occupy more memory space than values of type `int`, and arithmetic operations on `int32` are generally slower than those on `int`. Use `int32` only when the application requires exact 32-bit arithmetic.

`val zero : int32`

The 32-bit integer 0.

`val one : int32`

The 32-bit integer 1.

`val minus_one : int32`

The 32-bit integer -1.

`val neg : int32 -> int32`

Unary negation.

`val add : int32 -> int32 -> int32`

Addition.

`val sub : int32 -> int32 -> int32`

Subtraction.

`val mul : int32 -> int32 -> int32`

Multiplication.

`val div : int32 -> int32 -> int32`

Integer division. Raise `Division_by_zero` if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for `Pervasives.(/)`[17].

`val rem : int32 -> int32 -> int32`

Integer remainder. If `y` is not zero, the result of `Int32.rem x y` satisfies the following property: `x = Int32.add (Int32.mul (Int32.div x y) y) (Int32.rem x y)`. If `y = 0`, `Int32.rem x y` raises `Division_by_zero`.

`val succ : int32 -> int32`

Successor. `Int32.succ x` is `Int32.add x Int32.one`.

`val pred : int32 -> int32`

Predecessor. `Int32.pred x` is `Int32.sub x Int32.one`.

`val abs : int32 -> int32`

Return the absolute value of its argument.

`val max_int : int32`

The greatest representable 32-bit integer, $2^{31} - 1$.

`val min_int : int32`

The smallest representable 32-bit integer, -2^{31} .

`val logand : int32 -> int32 -> int32`

Bitwise logical and.

`val logor : int32 -> int32 -> int32`

Bitwise logical or.

`val logxor : int32 -> int32 -> int32`

Bitwise logical exclusive or.

`val lognot : int32 -> int32`

Bitwise logical negation

`val shift_left : int32 -> int -> int32`

`Int32.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= 32`.

`val shift_right : int32 -> int -> int32`

`Int32.shift_right x y` shifts `x` to the right by `y` bits. This is an arithmetic shift: the sign bit of `x` is replicated and inserted in the vacated bits. The result is unspecified if `y < 0` or `y >= 32`.

`val shift_right_logical : int32 -> int -> int32`

`Int32.shift_right_logical x y` shifts `x` to the right by `y` bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of `x`. The result is unspecified if `y < 0` or `y >= 32`.

`val of_int : int -> int32`

Convert the given integer (type `int`) to a 32-bit integer (type `int32`).

`val to_int : int32 -> int`

Convert the given 32-bit integer (type `int32`) to an integer (type `int`). On 32-bit platforms, the 32-bit integer is taken modulo 2^{31} , i.e. the high-order bit is lost during the conversion. On 64-bit platforms, the conversion is exact.

`val of_float : float -> int32`

Convert the given floating-point number to a 32-bit integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range `[Int32.min_int[5], Int32.max_int[5]]`.

`val to_float : int32 -> float`

Convert the given 32-bit integer to a floating-point number.

`val of_string : string -> int32`

Convert the given string to a 32-bit integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer, or if the integer represented exceeds the range of integers representable in type `int32`.

`val to_string : int32 -> string`

Return the string representation of its argument, in signed decimal.

`val bits_of_float : float -> int32`

Return the internal representation of the given float according to the IEEE 754 floating-point 'single format' bit layout. Bit 31 of the result represents the sign of the float; bits 30 to 23 represent the (biased) exponent; bits 22 to 0 represent the mantissa.

`val float_of_bits : int32 -> float`

Return the floating-point number whose internal representation, according to the IEEE 754 floating-point 'single format' bit layout, is the given `int32`.

`type t = int32`

An alias for the type of 32-bit integers.

`val compare : t -> t -> int`

The comparison function for 32-bit integers, with the same specification as `Pervasives.compare`[17]. Along with the type `t`, this function `compare` allows the module `Int32` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

Deprecated functions

```
val format : string -> int32 -> string
```

Do not use this deprecated function. Instead, used `Printf.sprintf[24]` with a `%1...` format.

6 Module Int64 : 64-bit integers.

This module provides operations on the type `int64` of signed 64-bit integers. Unlike the built-in `int` type, the type `int64` is guaranteed to be exactly 64-bit wide on all platforms. All arithmetic operations over `int64` are taken modulo 2^{64}

Performance notice: values of type `int64` occupy more memory space than values of type `int`, and arithmetic operations on `int64` are generally slower than those on `int`. Use `int64` only when the application requires exact 64-bit arithmetic.

```
val zero : int64
```

The 64-bit integer 0.

```
val one : int64
```

The 64-bit integer 1.

```
val minus_one : int64
```

The 64-bit integer -1.

```
val neg : int64 -> int64
```

Unary negation.

```
val add : int64 -> int64 -> int64
```

Addition.

```
val sub : int64 -> int64 -> int64
```

Subtraction.

```
val mul : int64 -> int64 -> int64
```

Multiplication.

```
val div : int64 -> int64 -> int64
```

Integer division. Raise `Division_by_zero` if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for `Pervasives.(/)`[17].

```
val rem : int64 -> int64 -> int64
```

Integer remainder. If `y` is not zero, the result of `Int64.rem x y` satisfies the following property: `x = Int64.add (Int64.mul (Int64.div x y) y) (Int64.rem x y)`. If `y = 0`, `Int64.rem x y` raises `Division_by_zero`.

```
val succ : int64 -> int64
```

Successor. `Int64.succ x` is `Int64.add x Int64.one`.

`val pred : int64 -> int64`
 Predecessor. `Int64.pred x` is `Int64.sub x Int64.one`.

`val abs : int64 -> int64`
 Return the absolute value of its argument.

`val max_int : int64`
 The greatest representable 64-bit integer, $2^{63} - 1$.

`val min_int : int64`
 The smallest representable 64-bit integer, -2^{63} .

`val logand : int64 -> int64 -> int64`
 Bitwise logical and.

`val logor : int64 -> int64 -> int64`
 Bitwise logical or.

`val logxor : int64 -> int64 -> int64`
 Bitwise logical exclusive or.

`val lognot : int64 -> int64`
 Bitwise logical negation

`val shift_left : int64 -> int -> int64`
`Int64.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= 64`.

`val shift_right : int64 -> int -> int64`
`Int64.shift_right x y` shifts `x` to the right by `y` bits. This is an arithmetic shift: the sign bit of `x` is replicated and inserted in the vacated bits. The result is unspecified if `y < 0` or `y >= 64`.

`val shift_right_logical : int64 -> int -> int64`
`Int64.shift_right_logical x y` shifts `x` to the right by `y` bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of `x`. The result is unspecified if `y < 0` or `y >= 64`.

`val of_int : int -> int64`
 Convert the given integer (type `int`) to a 64-bit integer (type `int64`).

`val to_int : int64 -> int`

Convert the given 64-bit integer (type `int64`) to an integer (type `int`). On 64-bit platforms, the 64-bit integer is taken modulo 2^{63} , i.e. the high-order bit is lost during the conversion. On 32-bit platforms, the 64-bit integer is taken modulo 2^{31} , i.e. the top 33 bits are lost during the conversion.

`val of_float : float -> int64`

Convert the given floating-point number to a 64-bit integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range `[Int64.min_int[6], Int64.max_int[6]]`.

`val to_float : int64 -> float`

Convert the given 64-bit integer to a floating-point number.

`val of_int32 : int32 -> int64`

Convert the given 32-bit integer (type `int32`) to a 64-bit integer (type `int64`).

`val to_int32 : int64 -> int32`

Convert the given 64-bit integer (type `int64`) to a 32-bit integer (type `int32`). The 64-bit integer is taken modulo 2^{32} , i.e. the top 32 bits are lost during the conversion.

`val of_nativeint : nativeint -> int64`

Convert the given native integer (type `nativeint`) to a 64-bit integer (type `int64`).

`val to_nativeint : int64 -> nativeint`

Convert the given 64-bit integer (type `int64`) to a native integer. On 32-bit platforms, the 64-bit integer is taken modulo 2^{32} . On 64-bit platforms, the conversion is exact.

`val of_string : string -> int64`

Convert the given string to a 64-bit integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer, or if the integer represented exceeds the range of integers representable in type `int64`.

`val to_string : int64 -> string`

Return the string representation of its argument, in decimal.

`val bits_of_float : float -> int64`

Return the internal representation of the given float according to the IEEE 754 floating-point 'double format' bit layout. Bit 63 of the result represents the sign of the float; bits 62 to 52 represent the (biased) exponent; bits 51 to 0 represent the mantissa.

`val float_of_bits : int64 -> float`

Return the floating-point number whose internal representation, according to the IEEE 754 floating-point 'double format' bit layout, is the given `int64`.

```
type t = int64
```

An alias for the type of 64-bit integers.

```
val compare : t -> t -> int
```

The comparison function for 64-bit integers, with the same specification as `Pervasives.compare`[17]. Along with the type `t`, this function `compare` allows the module `Int64` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

Deprecated functions

```
val format : string -> int64 -> string
```

Do not use this deprecated function. Instead, used `Printf.sprintf`[24] with a `%L...` format.

7 Module Array : 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`[7].

```
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[7]`.

`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`.

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

Same as `Array.iter`[7], but the function is applied to the index of the element as first argument, and the element itself as second argument.

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

Same as `Array.map`[7], 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`.

`val make_float : int -> float array`

`Array.make_float n` returns a fresh float array of length `n`, with uninitialized data.

Since: 4.02

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, `Pervasives.compare`[17] 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`[7], 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`[7].

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

Same as `Array.sort`[7] or `Array.stable_sort`[7], whichever is faster on typical input.

Undocumented functions

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

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

8 Module Stack : Last-in first-out stacks.

This module implements stacks (LIFOs), with in-place modification.

```
type 'a t
```

The type of stacks containing elements of type `'a`.

```
exception Empty
```

Raised when `Stack.pop`[8] or `Stack.top`[8] is applied to an empty stack.

```
val create : unit -> 'a t
```

Return a new stack, initially empty.

```

val push : 'a -> 'a t -> unit
    push x s adds the element x at the top of stack s.

val pop : 'a t -> 'a
    pop s removes and returns the topmost element in stack s, or raises Empty if the stack is
    empty.

val top : 'a t -> 'a
    top s returns the topmost element in stack s, or raises Empty if the stack is empty.

val clear : 'a t -> unit
    Discard all elements from a stack.

val copy : 'a t -> 'a t
    Return a copy of the given stack.

val is_empty : 'a t -> bool
    Return true if the given stack is empty, false otherwise.

val length : 'a t -> int
    Return the number of elements in a stack.

val iter : ('a -> unit) -> 'a t -> unit
    iter f s applies f in turn to all elements of s, from the element at the top of the stack to
    the element at the bottom of the stack. The stack itself is unchanged.

```

9 Module **ArrayLabels** : Array operations.

```

val length : 'a array -> int
    Return the length (number of elements) of the given array.

val get : 'a array -> int -> 'a
    ArrayLabels.get a n returns the element number n of array a. The first element has
    number 0. The last element has number ArrayLabels.length a - 1. You can also write
    a.(n) instead of ArrayLabels.get a n.

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

val set : 'a array -> int -> 'a -> unit
    ArrayLabels.set a n x modifies array a in place, replacing element number n with x. You
    can also write a.(n) <- x instead of ArrayLabels.set a n x.

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

```

```

val make : int -> 'a -> 'a array
    ArrayLabels.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. ArrayLabels.create is an alias for ArrayLabels.make[9].

val init : int -> f:(int -> 'a) -> 'a array
    ArrayLabels.init n f returns a fresh array of length n, with element number i initialized
    to the result of f i. In other terms, ArrayLabels.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 : dimx:int -> dimy:int -> 'a -> 'a array array
    ArrayLabels.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 : dimx:int -> dimy:int -> 'a -> 'a array array
    Deprecated. ArrayLabels.create_matrix is an alias for ArrayLabels.make_matrix[9].

val append : 'a array -> 'a array -> 'a array
    ArrayLabels.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 ArrayLabels.append, but concatenates a list of arrays.

val sub : 'a array -> pos:int -> len:int -> 'a array
    ArrayLabels.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 > ArrayLabels.length a.

val copy : 'a array -> 'a array

```

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

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

`ArrayLabels.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 :`

`src:'a array -> src_pos:int -> dst:'a array -> dst_pos:int -> len:int -> unit`

`ArrayLabels.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`

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

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

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

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

`ArrayLabels.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.(ArrayLabels.length a - 1); ()`.

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

`ArrayLabels.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.(ArrayLabels.length a - 1) |]`.

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

Same as `ArrayLabels.iter`[9], but the function is applied to the index of the element as first argument, and the element itself as second argument.

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

Same as `ArrayLabels.map`[9], but the function is applied to the index of the element as first argument, and the element itself as second argument.

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

`ArrayLabels.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 : f:('b -> 'a -> 'a) -> 'b array -> init:'a -> 'a`

`ArrayLabels.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`.

Sorting

```
val sort : cmp:('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, `Pervasives.compare`^[17] is a suitable comparison function, provided there are no floating-point NaN values in the data. After calling `ArrayLabels.sort`, the array is sorted in place in increasing order. `ArrayLabels.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 `ArrayLabels.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 : cmp:('a -> 'a -> int) -> 'a array -> unit
```

Same as `ArrayLabels.sort`^[9], 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 `ArrayLabels.sort`^[9].

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

Same as `ArrayLabels.sort`^[9] or `ArrayLabels.stable_sort`^[9], whichever is faster on typical input.

Undocumented functions

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

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

10 Module Digest : MD5 message digest.

This module provides functions to compute 128-bit 'digests' of arbitrary-length strings or files. The digests are of cryptographic quality: it is very hard, given a digest, to forge a string having that

digest. The algorithm used is MD5. This module should not be used for secure and sensitive cryptographic applications. For these kind of applications more recent and stronger cryptographic primitives should be used instead.

type `t` = `string`

The type of digests: 16-character strings.

val `compare` : `t` -> `t` -> `int`

The comparison function for 16-character digest, with the same specification as `Pervasives.compare`[17] and the implementation shared with `String.compare`[44]. Along with the type `t`, this function `compare` allows the module `Digest` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

Since: 4.00.0

val `string` : `string` -> `t`

Return the digest of the given string.

val `bytes` : `bytes` -> `t`

Return the digest of the given byte sequence.

Since: 4.02.0

val `substring` : `string` -> `int` -> `int` -> `t`

`Digest.substring s ofs len` returns the digest of the substring of `s` starting at index `ofs` and containing `len` characters.

val `subbytes` : `bytes` -> `int` -> `int` -> `t`

`Digest.subbytes s ofs len` returns the digest of the subsequence of `s` starting at index `ofs` and containing `len` bytes.

Since: 4.02.0

val `channel` : `Pervasives.in_channel` -> `int` -> `t`

If `len` is nonnegative, `Digest.channel ic len` reads `len` characters from channel `ic` and returns their digest, or raises `End_of_file` if end-of-file is reached before `len` characters are read. If `len` is negative, `Digest.channel ic len` reads all characters from `ic` until end-of-file is reached and return their digest.

val `file` : `string` -> `t`

Return the digest of the file whose name is given.

val `output` : `Pervasives.out_channel` -> `t` -> `unit`

Write a digest on the given output channel.

val `input` : `Pervasives.in_channel` -> `t`

Read a digest from the given input channel.

```
val to_hex : t -> string
```

Return the printable hexadecimal representation of the given digest.

```
val from_hex : string -> t
```

Convert a hexadecimal representation back into the corresponding digest. Raise `Invalid_argument` if the argument is not exactly 32 hexadecimal characters.

Since: 4.00.0

11 Module Complex : Complex numbers.

This module provides arithmetic operations on complex numbers. Complex numbers are represented by their real and imaginary parts (cartesian representation). Each part is represented by a double-precision floating-point number (type `float`).

```
type t = {  
  re : float ;  
  im : float ;  
}
```

The type of complex numbers. `re` is the real part and `im` the imaginary part.

```
val zero : t
```

The complex number 0.

```
val one : t
```

The complex number 1.

```
val i : t
```

The complex number i .

```
val neg : t -> t
```

Unary negation.

```
val conj : t -> t
```

Conjugate: given the complex $x + i.y$, returns $x - i.y$.

```
val add : t -> t -> t
```

Addition

```
val sub : t -> t -> t
```

Subtraction

```
val mul : t -> t -> t
```

Multiplication

```

val inv : t -> t
    Multiplicative inverse (1/z).

val div : t -> t -> t
    Division

val sqrt : t -> t
    Square root. The result  $x + i.y$  is such that  $x > 0$  or  $x = 0$  and  $y \geq 0$ . This function has
    a discontinuity along the negative real axis.

val norm2 : t -> float
    Norm squared: given  $x + i.y$ , returns  $x^2 + y^2$ .

val norm : t -> float
    Norm: given  $x + i.y$ , returns  $\text{sqrt}(x^2 + y^2)$ .

val arg : t -> float
    Argument. The argument of a complex number is the angle in the complex plane between
    the positive real axis and a line passing through zero and the number. This angle ranges
    from  $-\pi$  to  $\pi$ . This function has a discontinuity along the negative real axis.

val polar : float -> float -> t
    polar norm arg returns the complex having norm norm and argument arg.

val exp : t -> t
    Exponentiation. exp z returns  $e$  to the  $z$  power.

val log : t -> t
    Natural logarithm (in base  $e$ ).

val pow : t -> t -> t
    Power function. pow z1 z2 returns  $z1$  to the  $z2$  power.

```

12 Module Bytes : Byte sequence operations.

A byte sequence is a mutable data structure that contains a fixed-length sequence of bytes. Each byte can be indexed in constant time for reading or writing.

Given a byte sequence `s` of length `l`, we can access each of the `l` bytes of `s` via its index in the sequence. Indexes start at 0, and we will call an index valid in `s` if it falls within the range `[0...l-1]` (inclusive). A position is the point between two bytes or at the beginning or end of the sequence. We call a position valid in `s` if it falls within the range `[0...l]` (inclusive). Note that the byte at index `n` is between positions `n` and `n+1`.

Two parameters `start` and `len` are said to designate a valid range of `s` if `len` ≥ 0 and `start` and `start+len` are valid positions in `s`.

Byte sequences can be modified in place, for instance via the `set` and `blit` functions described below. See also strings (module `String`[44]), which are almost the same data structure, but cannot be modified in place.

Bytes are represented by the OCaml type `char`.

Since: 4.02.0

`val length : bytes -> int`

Return the length (number of bytes) of the argument.

`val get : bytes -> int -> char`

`get s n` returns the byte at index `n` in argument `s`.

Raise `Invalid_argument` if `n` not a valid index in `s`.

`val set : bytes -> int -> char -> unit`

`set s n c` modifies `s` in place, replacing the byte at index `n` with `c`.

Raise `Invalid_argument` if `n` is not a valid index in `s`.

`val create : int -> bytes`

`create n` returns a new byte sequence of length `n`. The sequence is uninitialized and contains arbitrary bytes.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val make : int -> char -> bytes`

`make n c` returns a new byte sequence of length `n`, filled with the byte `c`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val init : int -> (int -> char) -> bytes`

`Bytes.init n f` returns a fresh byte sequence of length `n`, with character `i` initialized to the result of `f i` (in increasing index order).

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val empty : bytes`

A byte sequence of size 0.

`val copy : bytes -> bytes`

Return a new byte sequence that contains the same bytes as the argument.

`val of_string : string -> bytes`

Return a new byte sequence that contains the same bytes as the given string.

`val to_string : bytes -> string`

Return a new string that contains the same bytes as the given byte sequence.

```

val sub : bytes -> int -> int -> bytes
    sub s start len returns a new byte sequence of length len, containing the subsequence of
    s that starts at position start and has length len.
    Raise Invalid_argument if start and len do not designate a valid range of s.

val sub_string : bytes -> int -> int -> string
    Same as sub but return a string instead of a byte sequence.

val extend : bytes -> int -> int -> bytes
    extend s left right returns a new byte sequence that contains the bytes of s, with left
    uninitialized bytes prepended and right uninitialized bytes appended to it. If left or right
    is negative, then bytes are removed (instead of appended) from the corresponding side of s.
    Raise Invalid_argument if the result length is negative or longer than
    Sys.max_string_length[22] bytes.

val fill : bytes -> int -> int -> char -> unit
    fill s start len c modifies s in place, replacing len characters with c, starting at start.
    Raise Invalid_argument if start and len do not designate a valid range of s.

val blit : bytes -> int -> bytes -> int -> int -> unit
    blit src srcoff dst dstoff len copies len bytes from sequence src, starting at index
    srcoff, to sequence dst, starting at index dstoff. It works correctly even if src and dst
    are the same byte sequence, and the source and destination intervals overlap.
    Raise Invalid_argument if srcoff and len do not designate a valid range of src, or if
    dstoff and len do not designate a valid range of dst.

val blit_string : string -> int -> bytes -> int -> int -> unit
    blit src srcoff dst dstoff len copies len bytes from string src, starting at index
    srcoff, to byte sequence dst, starting at index dstoff.
    Raise Invalid_argument if srcoff and len do not designate a valid range of src, or if
    dstoff and len do not designate a valid range of dst.

val concat : bytes -> bytes list -> bytes
    concat sep s1 concatenates the list of byte sequences s1, inserting the separator byte
    sequence sep between each, and returns the result as a new byte sequence.
    Raise Invalid_argument if the result is longer than Sys.max_string_length[22] bytes.

val cat : bytes -> bytes -> bytes
    cat s1 s2 concatenates s1 and s2 and returns the result as new byte sequence.
    Raise Invalid_argument if the result is longer than Sys.max_string_length[22] bytes.

val iter : (char -> unit) -> bytes -> unit

```

`iter f s` applies function `f` in turn to all the bytes of `s`. It is equivalent to `f (get s 0); f (get s 1); ...; f (get s (length s - 1)); ()`.

`val iteri : (int -> char -> unit) -> bytes -> unit`
 Same as `Bytes.iter[12]`, but the function is applied to the index of the byte as first argument and the byte itself as second argument.

`val map : (char -> char) -> bytes -> bytes`
`map f s` applies function `f` in turn to all the bytes of `s` (in increasing index order) and stores the resulting bytes in a new sequence that is returned as the result.

`val mapi : (int -> char -> char) -> bytes -> bytes`
`mapi f s` calls `f` with each character of `s` and its index (in increasing index order) and stores the resulting bytes in a new sequence that is returned as the result.

`val trim : bytes -> bytes`
 Return a copy of the argument, without leading and trailing whitespace. The bytes regarded as whitespace are the ASCII characters ' ', '\012', '\n', '\r', and '\t'.

`val escaped : bytes -> bytes`
 Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of OCaml.
 Raise `Invalid_argument` if the result is longer than `Sys.max_string_length[22]` bytes.

`val index : bytes -> char -> int`
`index s c` returns the index of the first occurrence of byte `c` in `s`.
 Raise `Not_found` if `c` does not occur in `s`.

`val rindex : bytes -> char -> int`
`rindex s c` returns the index of the last occurrence of byte `c` in `s`.
 Raise `Not_found` if `c` does not occur in `s`.

`val index_from : bytes -> int -> char -> int`
`index_from s i c` returns the index of the first occurrence of byte `c` in `s` after position `i`.
`Bytes.index s c` is equivalent to `Bytes.index_from s 0 c`.
 Raise `Invalid_argument` if `i` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` after position `i`.

`val rindex_from : bytes -> int -> char -> int`
`rindex_from s i c` returns the index of the last occurrence of byte `c` in `s` before position `i+1`. `rindex s c` is equivalent to `rindex_from s (Bytes.length s - 1) c`.
 Raise `Invalid_argument` if `i+1` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` before position `i+1`.

```

val contains : bytes -> char -> bool
    contains s c tests if byte c appears in s.

val contains_from : bytes -> int -> char -> bool
    contains_from s start c tests if byte c appears in s after position start. contains s c
    is equivalent to contains_from s 0 c.
    Raise Invalid_argument if start is not a valid position in s.

val rcontains_from : bytes -> int -> char -> bool
    rcontains_from s stop c tests if byte c appears in s before position stop+1.
    Raise Invalid_argument if stop < 0 or stop+1 is not a valid position in s.

val uppercase : bytes -> bytes
    Return a copy of the argument, with all lowercase letters translated to uppercase, including
    accented letters of the ISO Latin-1 (8859-1) character set.

val lowercase : bytes -> bytes
    Return a copy of the argument, with all uppercase letters translated to lowercase, including
    accented letters of the ISO Latin-1 (8859-1) character set.

val capitalize : bytes -> bytes
    Return a copy of the argument, with the first byte set to uppercase.

val uncapitalize : bytes -> bytes
    Return a copy of the argument, with the first byte set to lowercase.

type t = bytes
    An alias for the type of byte sequences.

val compare : t -> t -> int
    The comparison function for byte sequences, with the same specification as
    Pervasives.compare[17]. Along with the type t, this function compare allows the module
    Bytes to be passed as argument to the functors Set.Make[38] and Map.Make[3].

```

Unsafe conversions (for advanced users)

This section describes unsafe, low-level conversion functions between `bytes` and `string`. They do not copy the internal data; used improperly, they can break the immutability invariant on strings provided by the `-safe-string` option. They are available for expert library authors, but for most purposes you should use the always-correct `Bytes.to_string`[12] and `Bytes.of_string`[12] instead.

```

val unsafe_to_string : bytes -> string

```

Unsafely convert a byte sequence into a string.

To reason about the use of `unsafe_to_string`, it is convenient to consider an "ownership" discipline. A piece of code that manipulates some data "owns" it; there are several disjoint ownership modes, including:

- Unique ownership: the data may be accessed and mutated
- Shared ownership: the data has several owners, that may only access it, not mutate it.

Unique ownership is linear: passing the data to another piece of code means giving up ownership (we cannot write the data again). A unique owner may decide to make the data shared (giving up mutation rights on it), but shared data may not become uniquely-owned again.

`unsafe_to_string s` can only be used when the caller owns the byte sequence `s` – either uniquely or as shared immutable data. The caller gives up ownership of `s`, and gains ownership of the returned string.

There are two valid use-cases that respect this ownership discipline:

1. Creating a string by initializing and mutating a byte sequence that is never changed after initialization is performed.

```
let string_init len f : string =
  let s = Bytes.create len in
  for i = 0 to len - 1 do Bytes.set s i (f i) done;
  Bytes.unsafe_to_string s
```

This function is safe because the byte sequence `s` will never be accessed or mutated after `unsafe_to_string` is called. The `string_init` code gives up ownership of `s`, and returns the ownership of the resulting string to its caller.

Note that it would be unsafe if `s` was passed as an additional parameter to the function `f` as it could escape this way and be mutated in the future – `string_init` would give up ownership of `s` to pass it to `f`, and could not call `unsafe_to_string` safely.

We have provided the `String.init`^[44], `String.map`^[44] and `String.mapi`^[44] functions to cover most cases of building new strings. You should prefer those over `to_string` or `unsafe_to_string` whenever applicable.

2. Temporarily giving ownership of a byte sequence to a function that expects a uniquely owned string and returns ownership back, so that we can mutate the sequence again after the call ended.

```
let bytes_length (s : bytes) =
  String.length (Bytes.unsafe_to_string s)
```

In this use-case, we do not promise that `s` will never be mutated after the call to `bytes_length s`. The `String.length`^[44] function temporarily borrows unique ownership of the byte sequence (and sees it as a `string`), but returns this ownership back to the caller, which may assume that `s` is still a valid byte sequence after the call. Note that this is only correct because we know that `String.length`^[44] does not capture its argument – it could escape by a side-channel such as a memoization combinator.

The caller may not mutate `s` while the string is borrowed (it has temporarily given up ownership). This affects concurrent programs, but also higher-order functions: if

`String.length` returned a closure to be called later, `s` should not be mutated until this closure is fully applied and returns ownership.

```
val unsafe_of_string : string -> bytes
```

Unsafely convert a shared string to a byte sequence that should not be mutated.

The same ownership discipline that makes `unsafe_to_string` correct applies to `unsafe_of_string`: you may use it if you were the owner of the `string` value, and you will own the return `bytes` in the same mode.

In practice, unique ownership of string values is extremely difficult to reason about correctly. You should always assume strings are shared, never uniquely owned.

For example, string literals are implicitly shared by the compiler, so you never uniquely own them.

```
let incorrect = Bytes.unsafe_of_string "hello"
let s = Bytes.of_string "hello"
```

The first declaration is incorrect, because the string literal `"hello"` could be shared by the compiler with other parts of the program, and mutating `incorrect` is a bug. You must always use the second version, which performs a copy and is thus correct.

Assuming unique ownership of strings that are not string literals, but are (partly) built from string literals, is also incorrect. For example, mutating `unsafe_of_string ("foo" ^ s)` could mutate the shared string `"foo"` – assuming a rope-like representation of strings. More generally, functions operating on strings will assume shared ownership, they do not preserve unique ownership. It is thus incorrect to assume unique ownership of the result of `unsafe_of_string`.

The only case we have reasonable confidence is safe is if the produced `bytes` is shared – used as an immutable byte sequence. This is possibly useful for incremental migration of low-level programs that manipulate immutable sequences of bytes (for example `Marshal.from_bytes[21]`) and previously used the `string` type for this purpose.

13 Module Parsing : The run-time library for parsers generated by ocaml yacc.

```
val symbol_start : unit -> int
```

`symbol_start` and `Parsing.symbol_end[13]` are to be called in the action part of a grammar rule only. They return the offset of the string that matches the left-hand side of the rule: `symbol_start()` returns the offset of the first character; `symbol_end()` returns the offset after the last character. The first character in a file is at offset 0.

```
val symbol_end : unit -> int
```

See `Parsing.symbol_start[13]`.

val rhs_start : int -> int

Same as `Parsing.symbol_start[13]` and `Parsing.symbol_end[13]`, but return the offset of the string matching the `nth` item on the right-hand side of the rule, where `n` is the integer parameter to `rhs_start` and `rhs_end`. `n` is 1 for the leftmost item.

val rhs_end : int -> int

See `Parsing.rhs_start[13]`.

val symbol_start_pos : unit -> Lexing.position

Same as `symbol_start`, but return a `position` instead of an offset.

val symbol_end_pos : unit -> Lexing.position

Same as `symbol_end`, but return a `position` instead of an offset.

val rhs_start_pos : int -> Lexing.position

Same as `rhs_start`, but return a `position` instead of an offset.

val rhs_end_pos : int -> Lexing.position

Same as `rhs_end`, but return a `position` instead of an offset.

val clear_parser : unit -> unit

Empty the parser stack. Call it just after a parsing function has returned, to remove all pointers from the parser stack to structures that were built by semantic actions during parsing. This is optional, but lowers the memory requirements of the programs.

exception Parse_error

Raised when a parser encounters a syntax error. Can also be raised from the action part of a grammar rule, to initiate error recovery.

val set_trace : bool -> bool

Control debugging support for `ocamlyacc`-generated parsers. After `Parsing.set_trace true`, the pushdown automaton that executes the parsers prints a trace of its actions (reading a token, shifting a state, reducing by a rule) on standard output. `Parsing.set_trace false` turns this debugging trace off. The boolean returned is the previous state of the trace flag.

Since: 3.11.0

14 Module BytesLabels : Byte sequence operations.

Since: 4.02.0

val length : bytes -> int

Return the length (number of bytes) of the argument.

val get : bytes -> int -> char

`get s n` returns the byte at index `n` in argument `s`.
 Raise `Invalid_argument` if `n` not a valid index in `s`.

`val set : bytes -> int -> char -> unit`
`set s n c` modifies `s` in place, replacing the byte at index `n` with `c`.
 Raise `Invalid_argument` if `n` is not a valid index in `s`.

`val create : int -> bytes`
`create n` returns a new byte sequence of length `n`. The sequence is uninitialized and contains arbitrary bytes.
 Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val make : int -> char -> bytes`
`make n c` returns a new byte sequence of length `n`, filled with the byte `c`.
 Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val init : int -> f:(int -> char) -> bytes`
`init n f` returns a fresh byte sequence of length `n`, with character `i` initialized to the result of `f i`.
 Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val empty : bytes`
 A byte sequence of size 0.

`val copy : bytes -> bytes`
 Return a new byte sequence that contains the same bytes as the argument.

`val of_string : string -> bytes`
 Return a new byte sequence that contains the same bytes as the given string.

`val to_string : bytes -> string`
 Return a new string that contains the same bytes as the given byte sequence.

`val sub : bytes -> pos:int -> len:int -> bytes`
`sub s start len` returns a new byte sequence of length `len`, containing the subsequence of `s` that starts at position `start` and has length `len`.
 Raise `Invalid_argument` if `start` and `len` do not designate a valid range of `s`.

`val sub_string : bytes -> int -> int -> string`
 Same as `sub` but return a string instead of a byte sequence.

`val fill : bytes -> pos:int -> len:int -> char -> unit`

`fill s start len c` modifies `s` in place, replacing `len` characters with `c`, starting at `start`.
Raise `Invalid_argument` if `start` and `len` do not designate a valid range of `s`.

`val blit :`

`src:bytes -> src_pos:int -> dst:bytes -> dst_pos:int -> len:int -> unit`

`blit src srcoff dst dstoff len` copies `len` bytes from sequence `src`, starting at index `srcoff`, to sequence `dst`, starting at index `dstoff`. It works correctly even if `src` and `dst` are the same byte sequence, and the source and destination intervals overlap.

Raise `Invalid_argument` if `srcoff` and `len` do not designate a valid range of `src`, or if `dstoff` and `len` do not designate a valid range of `dst`.

`val concat : sep:bytes -> bytes list -> bytes`

`concat sep sl` concatenates the list of byte sequences `sl`, inserting the separator byte sequence `sep` between each, and returns the result as a new byte sequence.

`val iter : f:(char -> unit) -> bytes -> unit`

`iter f s` applies function `f` in turn to all the bytes of `s`. It is equivalent to `f (get s 0); f (get s 1); ...; f (get s (length s - 1)); ()`.

`val iteri : f:(int -> char -> unit) -> bytes -> unit`

Same as `Bytes.iter[12]`, but the function is applied to the index of the byte as first argument and the byte itself as second argument.

`val map : f:(char -> char) -> bytes -> bytes`

`map f s` applies function `f` in turn to all the bytes of `s` and stores the resulting bytes in a new sequence that is returned as the result.

`val mapi : f:(int -> char -> char) -> bytes -> bytes`

`mapi f s` calls `f` with each character of `s` and its index (in increasing index order) and stores the resulting bytes in a new sequence that is returned as the result.

`val trim : bytes -> bytes`

Return a copy of the argument, without leading and trailing whitespace. The bytes regarded as whitespace are the ASCII characters ' ', '\012', '\n', '\r', and '\t'.

`val escaped : bytes -> bytes`

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of OCaml.

`val index : bytes -> char -> int`

`index s c` returns the index of the first occurrence of byte `c` in `s`.

Raise `Not_found` if `c` does not occur in `s`.

`val rindex : bytes -> char -> int`

`rindex s c` returns the index of the last occurrence of byte `c` in `s`.
 Raise `Not_found` if `c` does not occur in `s`.

`val index_from : bytes -> int -> char -> int`
`index_from s i c` returns the index of the first occurrence of byte `c` in `s` after position `i`.
`Bytes.index s c` is equivalent to `Bytes.index_from s 0 c`.
 Raise `Invalid_argument` if `i` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` after position `i`.

`val rindex_from : bytes -> int -> char -> int`
`rindex_from s i c` returns the index of the last occurrence of byte `c` in `s` before position `i+1`. `rindex s c` is equivalent to `rindex_from s (Bytes.length s - 1) c`.
 Raise `Invalid_argument` if `i+1` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` before position `i+1`.

`val contains : bytes -> char -> bool`
`contains s c` tests if byte `c` appears in `s`.

`val contains_from : bytes -> int -> char -> bool`
`contains_from s start c` tests if byte `c` appears in `s` after position `start`. `contains s c` is equivalent to `contains_from s 0 c`.
 Raise `Invalid_argument` if `start` is not a valid position in `s`.

`val rcontains_from : bytes -> int -> char -> bool`
`rcontains_from s stop c` tests if byte `c` appears in `s` before position `stop+1`.
 Raise `Invalid_argument` if `stop < 0` or `stop+1` is not a valid position in `s`.

`val uppercase : bytes -> bytes`
 Return a copy of the argument, with all lowercase letters translated to uppercase, including accented letters of the ISO Latin-1 (8859-1) character set.

`val lowercase : bytes -> bytes`
 Return a copy of the argument, with all uppercase letters translated to lowercase, including accented letters of the ISO Latin-1 (8859-1) character set.

`val capitalize : bytes -> bytes`
 Return a copy of the argument, with the first byte set to uppercase.

`val uncapitalize : bytes -> bytes`
 Return a copy of the argument, with the first byte set to lowercase.

`type t = bytes`
 An alias for the type of byte sequences.

```
val compare : t -> t -> int
```

The comparison function for byte sequences, with the same specification as `Pervasives.compare`[17]. Along with the type `t`, this function `compare` allows the module `Bytes` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

15 Module Arg : Parsing of command line arguments.

This module provides a general mechanism for extracting options and arguments from the command line to the program.

Syntax of command lines: A keyword is a character string starting with a `-`. An option is a keyword alone or followed by an argument. The types of keywords are: `Unit`, `Bool`, `Set`, `Clear`, `String`, `Set_string`, `Int`, `Set_int`, `Float`, `Set_float`, `Tuple`, `Symbol`, and `Rest`. `Unit`, `Set` and `Clear` keywords take no argument. A `Rest` keyword takes the remaining of the command line as arguments. Every other keyword takes the following word on the command line as argument. Arguments not preceded by a keyword are called anonymous arguments.

Examples (`cmd` is assumed to be the command name):

- `cmd -flag` (a unit option)
- `cmd -int 1` (an int option with argument 1)
- `cmd -string foobar` (a string option with argument "foobar")
- `cmd -float 12.34` (a float option with argument 12.34)
- `cmd a b c` (three anonymous arguments: "a", "b", and "c")
- `cmd a b - c d` (two anonymous arguments and a rest option with two arguments)

```
type spec =  
  | Unit of (unit -> unit)  
    Call the function with unit argument  
  | Bool of (bool -> unit)  
    Call the function with a bool argument  
  | Set of bool Pervasives.ref  
    Set the reference to true  
  | Clear of bool Pervasives.ref  
    Set the reference to false  
  | String of (string -> unit)  
    Call the function with a string argument  
  | Set_string of string Pervasives.ref  
    Set the reference to the string argument
```

```

| Int of (int -> unit)
    Call the function with an int argument
| Set_int of int Pervasives.ref
    Set the reference to the int argument
| Float of (float -> unit)
    Call the function with a float argument
| Set_float of float Pervasives.ref
    Set the reference to the float argument
| Tuple of spec list
    Take several arguments according to the spec list
| Symbol of string list * (string -> unit)
    Take one of the symbols as argument and call the function with the symbol
| Rest of (string -> unit)
    Stop interpreting keywords and call the function with each remaining argument
    The concrete type describing the behavior associated with a keyword.

```

```

type key = string
type doc = string
type usage_msg = string
type anon_fun = string -> unit
val parse : (key * spec * doc) list -> anon_fun -> usage_msg -> unit

```

`Arg.parse speclist anon_fun usage_msg` parses the command line. `speclist` is a list of triples (`key`, `spec`, `doc`). `key` is the option keyword, it must start with a '-' character. `spec` gives the option type and the function to call when this option is found on the command line. `doc` is a one-line description of this option. `anon_fun` is called on anonymous arguments. The functions in `spec` and `anon_fun` are called in the same order as their arguments appear on the command line.

If an error occurs, `Arg.parse` exits the program, after printing to standard error an error message as follows:

- The reason for the error: unknown option, invalid or missing argument, etc.
- `usage_msg`
- The list of options, each followed by the corresponding `doc` string. Beware: options that have an empty `doc` string will not be included in the list.

For the user to be able to specify anonymous arguments starting with a -, include for example `("-", String anon_fun, doc)` in `speclist`.

By default, `parse` recognizes two unit options, `-help` and `-help`, which will print to standard output `usage_msg` and the list of options, and exit the program. You can override this behaviour by specifying your own `-help` and `-help` options in `speclist`.

```

val parse_dynamic :
  (key * spec * doc) list Pervasives.ref ->
  anon_fun -> usage_msg -> unit

```

Same as `Arg.parse[15]`, except that the `speclist` argument is a reference and may be updated during the parsing. A typical use for this feature is to parse command lines of the form:

- command subcommand `options` where the list of options depends on the value of the subcommand argument.

```

val parse_argv :
  ?current:int Pervasives.ref ->
  string array ->
  (key * spec * doc) list -> anon_fun -> usage_msg -> unit

```

`Arg.parse_argv ~current args speclist anon_fun usage_msg` parses the array `args` as if it were the command line. It uses and updates the value of `~current` (if given), or `Arg.current`. You must set it before calling `parse_argv`. The initial value of `current` is the index of the program name (argument 0) in the array. If an error occurs, `Arg.parse_argv` raises `Arg.Bad` with the error message as argument. If option `-help` or `-help` is given, `Arg.parse_argv` raises `Arg.Help` with the help message as argument.

```

val parse_argv_dynamic :
  ?current:int Pervasives.ref ->
  string array ->
  (key * spec * doc) list Pervasives.ref ->
  anon_fun -> string -> unit

```

Same as `Arg.parse_argv[15]`, except that the `speclist` argument is a reference and may be updated during the parsing. See `Arg.parse_dynamic[15]`.

exception Help of string

Raised by `Arg.parse_argv` when the user asks for help.

exception Bad of string

Functions in `spec` or `anon_fun` can raise `Arg.Bad` with an error message to reject invalid arguments. `Arg.Bad` is also raised by `Arg.parse_argv` in case of an error.

```

val usage : (key * spec * doc) list -> usage_msg -> unit

```

`Arg.usage speclist usage_msg` prints to standard error an error message that includes the list of valid options. This is the same message that `Arg.parse[15]` prints in case of error. `speclist` and `usage_msg` are the same as for `Arg.parse`.

```

val usage_string : (key * spec * doc) list -> usage_msg -> string

```

Returns the message that would have been printed by `Arg.usage[15]`, if provided with the same parameters.

```
val align :
  ?limit:int ->
  (key * spec * doc) list -> (key * spec * doc) list
```

Align the documentation strings by inserting spaces at the first space, according to the length of the keyword. Use a space as the first character in a doc string if you want to align the whole string. The doc strings corresponding to **Symbol** arguments are aligned on the next line.

```
val current : int Pervasives.ref
```

Position (in `Sys.argv[22]`) of the argument being processed. You can change this value, e.g. to force `Arg.parse[15]` to skip some arguments. `Arg.parse[15]` uses the initial value of `Arg.current[15]` as the index of argument 0 (the program name) and starts parsing arguments at the next element.

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. Raise **Invalid_argument** "List.nth" if *n* is negative.

```
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 concatenates it to `l2`. This is equivalent to `List.rev[16] 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

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

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[16]`, 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[16]`, 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[16] (List.map[16] 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 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 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[16] (List.map2[16] 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 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 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[16]`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists have different lengths.

`val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`
Same as `List.exists[16]`, 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 `List.mem[16]`, 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 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`[16].

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

Same as `List.assoc`[16], but uses physical equality instead of structural equality to compare keys.

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

Same as `List.assoc`[16], 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`[16], 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`[16], 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, `Pervasives.compare`^[17] 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`^[16], 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`^[16] or `List.stable_sort`^[16], whichever is faster on typical input.

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

Same as `List.sort`^[16], 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).

17 Module Pervasives : The initially opened module.

This module provides the basic operations over the built-in types (numbers, booleans, byte sequences, strings, exceptions, references, lists, arrays, input-output channels, ...).

This module is automatically opened at the beginning of each compilation. All components of this module can therefore be referred by their short name, without prefixing them by `Pervasives`.

Exceptions

```
val raise : exn -> 'a
```

Raise the given exception value

val `raise_notrace` : `exn -> 'a`

A faster version `raise` which does not record the backtrace.

Since: 4.02.0

val `invalid_arg` : `string -> 'a`

Raise exception `Invalid_argument` with the given string.

val `failwith` : `string -> 'a`

Raise exception `Failure` with the given string.

exception `Exit`

The `Exit` exception is not raised by any library function. It is provided for use in your programs.

Comparisons

val `(=)` : `'a -> 'a -> bool`

`e1 = e2` tests for structural equality of `e1` and `e2`. Mutable structures (e.g. references and arrays) are equal if and only if their current contents are structurally equal, even if the two mutable objects are not the same physical object. Equality between functional values raises `Invalid_argument`. Equality between cyclic data structures may not terminate.

val `(<>)` : `'a -> 'a -> bool`

Negation of `Pervasives.(=)`[17].

val `(<)` : `'a -> 'a -> bool`

See `Pervasives.(>=)`[17].

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

See `Pervasives.(>=)`[17].

val `(<=)` : `'a -> 'a -> bool`

See `Pervasives.(>=)`[17].

val `(>=)` : `'a -> 'a -> bool`

Structural ordering functions. These functions coincide with the usual orderings over integers, characters, strings, byte sequences and floating-point numbers, and extend them to a total ordering over all types. The ordering is compatible with `(=)`. As in the case of `(=)`, mutable structures are compared by contents. Comparison between functional values raises `Invalid_argument`. Comparison between cyclic structures may not terminate.

val `compare` : `'a -> 'a -> int`

`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`. The ordering implemented by `compare` is compatible with the comparison predicates `=`, `<` and `>` defined above, with one difference on the treatment of the float value `Pervasives.nan`[17]. Namely, the comparison predicates treat `nan` as different from any other float value, including itself; while `compare` treats `nan` as equal to itself and less than any other float value. This treatment of `nan` ensures that `compare` defines a total ordering relation.

`compare` applied to functional values may raise `Invalid_argument`. `compare` applied to cyclic structures may not terminate.

The `compare` function can be used as the comparison function required by the `Set.Make`[38] and `Map.Make`[3] functors, as well as the `List.sort`[16] and `Array.sort`[7] functions.

```
val min : 'a -> 'a -> 'a
```

Return the smaller of the two arguments. The result is unspecified if one of the arguments contains the float value `nan`.

```
val max : 'a -> 'a -> 'a
```

Return the greater of the two arguments. The result is unspecified if one of the arguments contains the float value `nan`.

```
val (==) : 'a -> 'a -> bool
```

`e1 == e2` tests for physical equality of `e1` and `e2`. On mutable types such as references, arrays, byte sequences, records with mutable fields and objects with mutable instance variables, `e1 == e2` is true if and only if physical modification of `e1` also affects `e2`. On non-mutable types, the behavior of `(==)` is implementation-dependent; however, it is guaranteed that `e1 == e2` implies `compare e1 e2 = 0`.

```
val (!=) : 'a -> 'a -> bool
```

Negation of `Pervasives.(==)`[17].

Boolean operations

```
val not : bool -> bool
```

The boolean negation.

```
val (&&) : bool -> bool -> bool
```

The boolean 'and'. Evaluation is sequential, left-to-right: in `e1 && e2`, `e1` is evaluated first, and if it returns `false`, `e2` is not evaluated at all.

```
val (&) : bool -> bool -> bool
```

Deprecated. `Pervasives.&&`[17] should be used instead.

```
val (||) : bool -> bool -> bool
```

The boolean 'or'. Evaluation is sequential, left-to-right: in `e1 || e2`, `e1` is evaluated first, and if it returns `true`, `e2` is not evaluated at all.

```

val (or) : bool -> bool -> bool
    Deprecated. Pervasives.(||)[17] should be used instead.

Debugging

val __LOC__ : string
    __LOC__ returns the location at which this expression appears in the file currently being
    parsed by the compiler, with the standard error format of OCaml: "File %S, line %d,
    characters %d-%d".
    Since: 4.02.0

val __FILE__ : string
    __FILE__ returns the name of the file currently being parsed by the compiler.
    Since: 4.02.0

val __LINE__ : int
    __LINE__ returns the line number at which this expression appears in the file currently being
    parsed by the compiler.
    Since: 4.02.0

val __MODULE__ : string
    __MODULE__ returns the module name of the file being parsed by the compiler.
    Since: 4.02.0

val __POS__ : string * int * int * int
    __POS__ returns a tuple (file, lnum, cnum, enum), corresponding to the location at which
    this expression appears in the file currently being parsed by the compiler. file is the
    current filename, lnum the line number, cnum the character position in the line and enum the
    last character position in the line.
    Since: 4.02.0

val __LOC_OF__ : 'a -> string * 'a
    __LOC_OF__ expr returns a pair (loc, expr) where loc is the location of expr in the file
    currently being parsed by the compiler, with the standard error format of OCaml: "File %S,
    line %d, characters %d-%d".
    Since: 4.02.0

val __LINE_OF__ : 'a -> int * 'a
    __LINE__ expr returns a pair (line, expr), where line is the line number at which the
    expression expr appears in the file currently being parsed by the compiler.
    Since: 4.02.0

val __POS_OF__ : 'a -> (string * int * int * int) * 'a

```

`__POS_OF__ expr` returns a pair `(loc,expr)`, where `loc` is a tuple `(file,lnum,cnum,enum)` corresponding to the location at which the expression `expr` appears in the file currently being parsed by the compiler. `file` is the current filename, `lnum` the line number, `cnum` the character position in the line and `enum` the last character position in the line.

Since: 4.02.0

Composition operators

`val (|>) : 'a -> ('a -> 'b) -> 'b`

Reverse-application operator: `x |> f |> g` is exactly equivalent to `g (f (x))`.

Since: 4.01

`val (@@) : ('a -> 'b) -> 'a -> 'b`

Application operator: `g @@ f @@ x` is exactly equivalent to `g (f (x))`.

Since: 4.01

Integer arithmetic

Integer arithmetic

Integers are 31 bits wide (or 63 bits on 64-bit processors). All operations are taken modulo 2^{31} (or 2^{63}). They do not fail on overflow.

`val (~-) : int -> int`

Unary negation. You can also write `- e` instead of `~- e`.

`val (~+) : int -> int`

Unary addition. You can also write `+ e` instead of `~+ e`.

Since: 3.12.0

`val succ : int -> int`

`succ x` is `x + 1`.

`val pred : int -> int`

`pred x` is `x - 1`.

`val (+) : int -> int -> int`

Integer addition.

`val (-) : int -> int -> int`

Integer subtraction.

`val (*) : int -> int -> int`

Integer multiplication.

`val (/) : int -> int -> int`

Integer division. Raise `Division_by_zero` if the second argument is 0. Integer division rounds the real quotient of its arguments towards zero. More precisely, if $x \geq 0$ and $y > 0$, x / y is the greatest integer less than or equal to the real quotient of x by y . Moreover, $(-x) / y = x / (-y) = -(x / y)$.

`val (mod) : int -> int -> int`

Integer remainder. If y is not zero, the result of `x mod y` satisfies the following properties: $x = (x / y) * y + x \bmod y$ and $\text{abs}(x \bmod y) \leq \text{abs}(y) - 1$. If $y = 0$, `x mod y` raises `Division_by_zero`. Note that `x mod y` is negative only if $x < 0$. Raise `Division_by_zero` if y is zero.

`val abs : int -> int`

Return the absolute value of the argument. Note that this may be negative if the argument is `min_int`.

`val max_int : int`

The greatest representable integer.

`val min_int : int`

The smallest representable integer.

Bitwise operations

`val (land) : int -> int -> int`

Bitwise logical and.

`val (lor) : int -> int -> int`

Bitwise logical or.

`val (lxor) : int -> int -> int`

Bitwise logical exclusive or.

`val lnot : int -> int`

Bitwise logical negation.

`val (lsl) : int -> int -> int`

`n lsl m` shifts `n` to the left by `m` bits. The result is unspecified if $m < 0$ or $m \geq \text{bitsize}$, where `bitsize` is 32 on a 32-bit platform and 64 on a 64-bit platform.

`val (lsr) : int -> int -> int`

`n lsr m` shifts `n` to the right by `m` bits. This is a logical shift: zeroes are inserted regardless of the sign of `n`. The result is unspecified if $m < 0$ or $m \geq \text{bitsize}$.

`val (asr) : int -> int -> int`

`n asr m` shifts `n` to the right by `m` bits. This is an arithmetic shift: the sign bit of `n` is replicated. The result is unspecified if $m < 0$ or $m \geq \text{bitsize}$.

Floating-point arithmetic

OCaml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers. Floating-point operations never raise an exception on overflow, underflow, division by zero, etc. Instead, special IEEE numbers are returned as appropriate, such as `infinity` for `1.0 /. 0.0`, `neg_infinity` for `-1.0 /. 0.0`, and `nan` ('not a number') for `0.0 /. 0.0`. These special numbers then propagate through floating-point computations as expected: for instance, `1.0 /. infinity` is `0.0`, and any arithmetic operation with `nan` as argument returns `nan` as result.

`val (~-.) : float -> float`

Unary negation. You can also write `-.` `e` instead of `~-.` `e`.

`val (~+.) : float -> float`

Unary addition. You can also write `+` `e` instead of `~+.` `e`.

Since: 3.12.0

`val (+.) : float -> float -> float`

Floating-point addition

`val (-.) : float -> float -> float`

Floating-point subtraction

`val (*.) : float -> float -> float`

Floating-point multiplication

`val (/.) : float -> float -> float`

Floating-point division.

`val (**) : float -> float -> float`

Exponentiation.

`val sqrt : float -> float`

Square root.

`val exp : float -> float`

Exponential.

`val log : float -> float`

Natural logarithm.

`val log10 : float -> float`

Base 10 logarithm.

`val expm1 : float -> float`

`expm1 x` computes `exp x -. 1.0`, giving numerically-accurate results even if `x` is close to `0.0`.

Since: 3.12.0

```

val log1p : float -> float
    log1p x computes  $\log(1.0 + x)$  (natural logarithm), giving numerically-accurate results
    even if x is close to 0.0.
    Since: 3.12.0

val cos : float -> float
    Cosine. Argument is in radians.

val sin : float -> float
    Sine. Argument is in radians.

val tan : float -> float
    Tangent. Argument is in radians.

val acos : float -> float
    Arc cosine. The argument must fall within the range  $[-1.0, 1.0]$ . Result is in radians and
    is between 0.0 and  $\pi$ .

val asin : float -> float
    Arc sine. The argument must fall within the range  $[-1.0, 1.0]$ . Result is in radians and is
    between  $-\pi/2$  and  $\pi/2$ .

val atan : float -> float
    Arc tangent. Result is in radians and is between  $-\pi/2$  and  $\pi/2$ .

val atan2 : float -> float -> float
    atan2 y x returns the arc tangent of  $y / x$ . The signs of x and y are used to determine
    the quadrant of the result. Result is in radians and is between  $-\pi$  and  $\pi$ .

val hypot : float -> float -> float
    hypot x y returns  $\sqrt{x^2 + y^2}$ , that is, the length of the hypotenuse of a
    right-angled triangle with sides of length x and y, or, equivalently, the distance of the point
    (x,y) to origin.
    Since: 4.00.0

val cosh : float -> float
    Hyperbolic cosine. Argument is in radians.

val sinh : float -> float
    Hyperbolic sine. Argument is in radians.

val tanh : float -> float
    Hyperbolic tangent. Argument is in radians.

```

```

val ceil : float -> float
    Round above to an integer value. ceil f returns the least integer value greater than or
    equal to f. The result is returned as a float.

val floor : float -> float
    Round below to an integer value. floor f returns the greatest integer value less than or
    equal to f. The result is returned as a float.

val abs_float : float -> float
    abs_float f returns the absolute value of f.

val copysign : float -> float -> float
    copysign x y returns a float whose absolute value is that of x and whose sign is that of y. If
    x is nan, returns nan. If y is nan, returns either x or -x, but it is not specified which.
    Since: 4.00.0

val mod_float : float -> float -> float
    mod_float a b returns the remainder of a with respect to b. The returned value is a - n * b,
    where n is the quotient a / b rounded towards zero to an integer.

val frexp : float -> float * int
    frexp f returns the pair of the significant and the exponent of f. When f is zero, the
    significant x and the exponent n of f are equal to zero. When f is non-zero, they are defined
    by f = x * 2 ** n and 0.5 <= x < 1.0.

val ldexp : float -> int -> float
    ldexp x n returns x * 2 ** n.

val modf : float -> float * float
    modf f returns the pair of the fractional and integral part of f.

val float : int -> float
    Same as Pervasives.float_of_int[17].

val float_of_int : int -> float
    Convert an integer to floating-point.

val truncate : float -> int
    Same as Pervasives.int_of_float[17].

val int_of_float : float -> int
    Truncate the given floating-point number to an integer. The result is unspecified if the
    argument is nan or falls outside the range of representable integers.

val infinity : float

```

Positive infinity.

```
val neg_infinity : float
```

Negative infinity.

```
val nan : float
```

A special floating-point value denoting the result of an undefined operation such as `0.0 /. 0.0`. Stands for 'not a number'. Any floating-point operation with `nan` as argument returns `nan` as result. As for floating-point comparisons, `=`, `<`, `<=`, `>` and `>=` return `false` and `<>` returns `true` if one or both of their arguments is `nan`.

```
val max_float : float
```

The largest positive finite value of type `float`.

```
val min_float : float
```

The smallest positive, non-zero, non-denormalized value of type `float`.

```
val epsilon_float : float
```

The difference between `1.0` and the smallest exactly representable floating-point number greater than `1.0`.

```
type fpclass =
```

```
| FP_normal
```

Normal number, none of the below

```
| FP_subnormal
```

Number very close to `0.0`, has reduced precision

```
| FP_zero
```

Number is `0.0` or `-0.0`

```
| FP_infinite
```

Number is positive or negative infinity

```
| FP_nan
```

Not a number: result of an undefined operation

The five classes of floating-point numbers, as determined by the `Pervasives.classify_float[17]` function.

```
val classify_float : float -> fpclass
```

Return the class of the given floating-point number: normal, subnormal, zero, infinite, or not a number.

String operations

More string operations are provided in module `String`[44].

```
val (^) : string -> string -> string
```

String concatenation.

Character operations

More character operations are provided in module `Char`[42].

`val int_of_char : char -> int`

Return the ASCII code of the argument.

`val char_of_int : int -> char`

Return the character with the given ASCII code. Raise `Invalid_argument "char_of_int"` if the argument is outside the range 0–255.

Unit operations

`val ignore : 'a -> unit`

Discard the value of its argument and return `()`. For instance, `ignore(f x)` discards the result of the side-effecting function `f`. It is equivalent to `f x; ()`, except that the latter may generate a compiler warning; writing `ignore(f x)` instead avoids the warning.

String conversion functions

`val string_of_bool : bool -> string`

Return the string representation of a boolean. As the returned values may be shared, the user should not modify them directly.

`val bool_of_string : string -> bool`

Convert the given string to a boolean. Raise `Invalid_argument "bool_of_string"` if the string is not `"true"` or `"false"`.

`val string_of_int : int -> string`

Return the string representation of an integer, in decimal.

`val int_of_string : string -> int`

Convert the given string to an integer. The string is read in decimal (by default) or in hexadecimal (if it begins with `0x` or `0X`), octal (if it begins with `0o` or `0O`), or binary (if it begins with `0b` or `0B`). Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer, or if the integer represented exceeds the range of integers representable in type `int`.

`val string_of_float : float -> string`

Return the string representation of a floating-point number.

`val float_of_string : string -> float`

Convert the given string to a float. Raise `Failure "float_of_string"` if the given string is not a valid representation of a float.

Pair operations

`val fst : 'a * 'b -> 'a`

Return the first component of a pair.

```
val snd : 'a * 'b -> 'b
```

Return the second component of a pair.

List operations

More list operations are provided in module `List[16]`.

```
val (@) : 'a list -> 'a list -> 'a list
```

List concatenation.

Input/output Note: all input/output functions can raise `Sys_error` when the system calls they invoke fail.

```
type in_channel
```

The type of input channel.

```
type out_channel
```

The type of output channel.

```
val stdin : in_channel
```

The standard input for the process.

```
val stdout : out_channel
```

The standard output for the process.

```
val stderr : out_channel
```

The standard error output for the process.

Output functions on standard output

```
val print_char : char -> unit
```

Print a character on standard output.

```
val print_string : string -> unit
```

Print a string on standard output.

```
val print_bytes : bytes -> unit
```

Print a byte sequence on standard output.

Since: 4.02.0

```
val print_int : int -> unit
```

Print an integer, in decimal, on standard output.

```
val print_float : float -> unit
```

Print a floating-point number, in decimal, on standard output.

```
val print_endline : string -> unit
```

Print a string, followed by a newline character, on standard output and flush standard output.

val print_newline : unit -> unit

Print a newline character on standard output, and flush standard output. This can be used to simulate line buffering of standard output.

Output functions on standard error

val prerr_char : char -> unit

Print a character on standard error.

val prerr_string : string -> unit

Print a string on standard error.

val prerr_bytes : bytes -> unit

Print a byte sequence on standard error.

Since: 4.02.0

val prerr_int : int -> unit

Print an integer, in decimal, on standard error.

val prerr_float : float -> unit

Print a floating-point number, in decimal, on standard error.

val prerr_endline : string -> unit

Print a string, followed by a newline character on standard error and flush standard error.

val prerr_newline : unit -> unit

Print a newline character on standard error, and flush standard error.

Input functions on standard input

val read_line : unit -> string

Flush standard output, then read characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

val read_int : unit -> int

Flush standard output, then read one line from standard input and convert it to an integer. Raise **Failure "int_of_string"** if the line read is not a valid representation of an integer.

val read_float : unit -> float

Flush standard output, then read one line from standard input and convert it to a floating-point number. The result is unspecified if the line read is not a valid representation of a floating-point number.

General output functions

```
type open_flag =  
  | Open_rdonly  
      open for reading.  
  | Open_wronly  
      open for writing.  
  | Open_append  
      open for appending: always write at end of file.  
  | Open_creat  
      create the file if it does not exist.  
  | Open_trunc  
      empty the file if it already exists.  
  | Open_excl  
      fail if Open_creat and the file already exists.  
  | Open_binary  
      open in binary mode (no conversion).  
  | Open_text  
      open in text mode (may perform conversions).  
  | Open_nonblock  
      open in non-blocking mode.  
Opening modes for Pervasives.open_out_gen[17] and Pervasives.open_in_gen[17].
```

```
val open_out : string -> out_channel
```

Open the named file for writing, and return a new output channel on that file, positionned at the beginning of the file. The file is truncated to zero length if it already exists. It is created if it does not already exists.

```
val open_out_bin : string -> out_channel
```

Same as `Pervasives.open_out[17]`, but the file is opened in binary mode, so that no translation takes place during writes. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `Pervasives.open_out[17]`.

```
val open_out_gen : open_flag list -> int -> string -> out_channel
```

`open_out_gen mode perm filename` opens the named file for writing, as described above. The extra argument `mode` specify the opening mode. The extra argument `perm` specifies the file permissions, in case the file must be created. `Pervasives.open_out[17]` and `Pervasives.open_out_bin[17]` are special cases of this function.

```
val flush : out_channel -> unit
```

Flush the buffer associated with the given output channel, performing all pending writes on that channel. Interactive programs must be careful about flushing standard output and standard error at the right time.

val flush_all : unit -> unit

Flush all open output channels; ignore errors.

val output_char : out_channel -> char -> unit

Write the character on the given output channel.

val output_string : out_channel -> string -> unit

Write the string on the given output channel.

val output_bytes : out_channel -> bytes -> unit

Write the byte sequence on the given output channel.

Since: 4.02.0

val output : out_channel -> bytes -> int -> int -> unit

`output oc buf pos len` writes `len` characters from byte sequence `buf`, starting at offset `pos`, to the given output channel `oc`. Raise `Invalid_argument "output"` if `pos` and `len` do not designate a valid range of `buf`.

val output_substring : out_channel -> string -> int -> int -> unit

Same as `output` but take a string as argument instead of a byte sequence.

Since: 4.02.0

val output_byte : out_channel -> int -> unit

Write one 8-bit integer (as the single character with that code) on the given output channel. The given integer is taken modulo 256.

val output_binary_int : out_channel -> int -> unit

Write one integer in binary format (4 bytes, big-endian) on the given output channel. The given integer is taken modulo 2^{32} . The only reliable way to read it back is through the `Pervasives.input_binary_int[17]` function. The format is compatible across all machines for a given version of OCaml.

val output_value : out_channel -> 'a -> unit

Write the representation of a structured value of any type to a channel. Circularities and sharing inside the value are detected and preserved. The object can be read back, by the function `Pervasives.input_value[17]`. See the description of module `Marshal`[21] for more information. `Pervasives.output_value[17]` is equivalent to `Marshal.to_channel[21]` with an empty list of flags.

val seek_out : out_channel -> int -> unit

`seek_out chan pos` sets the current writing position to `pos` for channel `chan`. This works only for regular files. On files of other kinds (such as terminals, pipes and sockets), the behavior is unspecified.

`val pos_out : out_channel -> int`

Return the current writing position for the given channel. Does not work on channels opened with the `Open_append` flag (returns unspecified results).

`val out_channel_length : out_channel -> int`

Return the size (number of characters) of the regular file on which the given channel is opened. If the channel is opened on a file that is not a regular file, the result is meaningless.

`val close_out : out_channel -> unit`

Close the given channel, flushing all buffered write operations. Output functions raise a `Sys_error` exception when they are applied to a closed output channel, except `close_out` and `flush`, which do nothing when applied to an already closed channel. Note that `close_out` may raise `Sys_error` if the operating system signals an error when flushing or closing.

`val close_out_noerr : out_channel -> unit`

Same as `close_out`, but ignore all errors.

`val set_binary_mode_out : out_channel -> bool -> unit`

`set_binary_mode_out oc true` sets the channel `oc` to binary mode: no translations take place during output. `set_binary_mode_out oc false` sets the channel `oc` to text mode: depending on the operating system, some translations may take place during output. For instance, under Windows, end-of-lines will be translated from `\n` to `\r\n`. This function has no effect under operating systems that do not distinguish between text mode and binary mode.

General input functions

`val open_in : string -> in_channel`

Open the named file for reading, and return a new input channel on that file, positioned at the beginning of the file.

`val open_in_bin : string -> in_channel`

Same as `Pervasives.open_in[17]`, but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `Pervasives.open_in[17]`.

`val open_in_gen : open_flag list -> int -> string -> in_channel`

`open_in_gen mode perm filename` opens the named file for reading, as described above.

The extra arguments `mode` and `perm` specify the opening mode and file permissions.

`Pervasives.open_in[17]` and `Pervasives.open_in_bin[17]` are special cases of this function.

`val input_char : in_channel -> char`

Read one character from the given input channel. Raise `End_of_file` if there are no more characters to read.

val `input_line` : `in_channel` -> `string`

Read characters from the given input channel, until a newline character is encountered. Return the string of all characters read, without the newline character at the end. Raise `End_of_file` if the end of the file is reached at the beginning of line.

val `input` : `in_channel` -> `bytes` -> `int` -> `int` -> `int`

`input ic buf pos len` reads up to `len` characters from the given channel `ic`, storing them in byte sequence `buf`, starting at character number `pos`. It returns the actual number of characters read, between 0 and `len` (inclusive). A return value of 0 means that the end of file was reached. A return value between 0 and `len` exclusive means that not all requested `len` characters were read, either because no more characters were available at that time, or because the implementation found it convenient to do a partial read; `input` must be called again to read the remaining characters, if desired. (See also `Pervasives.really_input`[17] for reading exactly `len` characters.) Exception `Invalid_argument "input"` is raised if `pos` and `len` do not designate a valid range of `buf`.

val `really_input` : `in_channel` -> `bytes` -> `int` -> `int` -> `unit`

`really_input ic buf pos len` reads `len` characters from channel `ic`, storing them in byte sequence `buf`, starting at character number `pos`. Raise `End_of_file` if the end of file is reached before `len` characters have been read. Raise `Invalid_argument "really_input"` if `pos` and `len` do not designate a valid range of `buf`.

val `really_input_string` : `in_channel` -> `int` -> `string`

`really_input_string ic len` reads `len` characters from channel `ic` and returns them in a new string. Raise `End_of_file` if the end of file is reached before `len` characters have been read.

Since: 4.02.0

val `input_byte` : `in_channel` -> `int`

Same as `Pervasives.input_char`[17], but return the 8-bit integer representing the character. Raise `End_of_file` if an end of file was reached.

val `input_binary_int` : `in_channel` -> `int`

Read an integer encoded in binary format (4 bytes, big-endian) from the given input channel. See `Pervasives.output_binary_int`[17]. Raise `End_of_file` if an end of file was reached while reading the integer.

val `input_value` : `in_channel` -> 'a

Read the representation of a structured value, as produced by `Pervasives.output_value`[17], and return the corresponding value. This function is identical to `Marshal.from_channel`[21]; see the description of module `Marshal`[21] for more information, in particular concerning the lack of type safety.

```

val seek_in : in_channel -> int -> unit
    seek_in chan pos sets the current reading position to pos for channel chan. This works
    only for regular files. On files of other kinds, the behavior is unspecified.

val pos_in : in_channel -> int
    Return the current reading position for the given channel.

val in_channel_length : in_channel -> int
    Return the size (number of characters) of the regular file on which the given channel is
    opened. If the channel is opened on a file that is not a regular file, the result is meaningless.
    The returned size does not take into account the end-of-line translations that can be
    performed when reading from a channel opened in text mode.

val close_in : in_channel -> unit
    Close the given channel. Input functions raise a Sys_error exception when they are applied
    to a closed input channel, except close_in, which does nothing when applied to an already
    closed channel.

val close_in_noerr : in_channel -> unit
    Same as close_in, but ignore all errors.

val set_binary_mode_in : in_channel -> bool -> unit
    set_binary_mode_in ic true sets the channel ic to binary mode: no translations take
    place during input. set_binary_mode_out ic false sets the channel ic to text mode:
    depending on the operating system, some translations may take place during input. For
    instance, under Windows, end-of-lines will be translated from \r\n to \n. This function has
    no effect under operating systems that do not distinguish between text mode and binary
    mode.

```

Operations on large files

```

module LargeFile :
sig
    val seek_out : Pervasives.out_channel -> int64 -> unit
    val pos_out : Pervasives.out_channel -> int64
    val out_channel_length : Pervasives.out_channel -> int64
    val seek_in : Pervasives.in_channel -> int64 -> unit
    val pos_in : Pervasives.in_channel -> int64
    val in_channel_length : Pervasives.in_channel -> int64
end

```

Operations on large files. This sub-module provides 64-bit variants of the channel functions that manipulate file positions and file sizes. By representing positions and sizes by 64-bit integers (type `int64`) instead of regular integers (type `int`), these alternate functions allow operating on files whose sizes are greater than `max_int`.

References

```
type 'a ref = {  
  mutable contents : 'a ;  
}
```

The type of references (mutable indirection cells) containing a value of type 'a.

```
val ref : 'a -> 'a ref
```

Return a fresh reference containing the given value.

```
val (!) : 'a ref -> 'a
```

`!r` returns the current contents of reference `r`. Equivalent to `fun r -> r.contents`.

```
val (:=) : 'a ref -> 'a -> unit
```

`r := a` stores the value of `a` in reference `r`. Equivalent to `fun r v -> r.contents <- v`.

```
val incr : int ref -> unit
```

Increment the integer contained in the given reference. Equivalent to `fun r -> r := succ !r`.

```
val decr : int ref -> unit
```

Decrement the integer contained in the given reference. Equivalent to `fun r -> r := pred !r`.

Operations on format strings

Operations on format strings

Format strings are character strings with special lexical conventions that defines the functionality of formatted input/output functions. Format strings are used to read data with formatted input functions from module `Scanf`[4] and to print data with formatted output functions from modules `Printf`[24] and `Format`[39].

Format strings are made of three kinds of entities:

- *conversions specifications*, introduced by the special character '%' followed by one or more characters specifying what kind of argument to read or print,
- *formatting indications*, introduced by the special character '@' followed by one or more characters specifying how to read or print the argument,
- *plain characters* that are regular characters with usual lexical conventions. Plain characters specify string literals to be read in the input or printed in the output.

There is an additional lexical rule to escape the special characters '%' and '@' in format strings: if a special character follows a '%' character, it is treated as a plain character. In other words, "%" is considered as a plain '%' and "%@" as a plain '@'.

For more information about conversion specifications and formatting indications available, read the documentation of modules `Scanf`[4], `Printf`[24] and `Format`[39].

Operations on format strings

Format strings are character strings with special lexical conventions that defines the functionality of formatted input/output functions. Format strings are used to read data with formatted input functions from module `Scanf`[4] and to print data with formatted output functions from modules `Printf`[24] and `Format`[39].

Format strings are made of three kinds of entities:

- *conversions specifications*, introduced by the special character '%' followed by one or more characters specifying what kind of argument to read or print,
- *formatting indications*, introduced by the special character '@' followed by one or more characters specifying how to read or print the argument,
- *plain characters* that are regular characters with usual lexical conventions. Plain characters specify string literals to be read in the input or printed in the output.

There is an additional lexical rule to escape the special characters '%' and '@' in format strings: if a special character follows a '%' character, it is treated as a plain character. In other words, "%" is considered as a plain '%' and "%@" as a plain '@'.

For more information about conversion specifications and formatting indications available, read the documentation of modules `Scanf`[4], `Printf`[24] and `Format`[39].

Format strings have a general and highly polymorphic type ('a, 'b, 'c, 'd, 'e, 'f) `format6`. The two simplified types, `format` and `format4` below are included for backward compatibility with earlier releases of OCaml.

The meaning of format string type parameters is as follows:

- 'a is the type of the parameters of the format for formatted output functions (`printf`-style functions); 'a is the type of the values read by the format for formatted input functions (`scanf`-style functions).
- 'b is the type of input source for formatted input functions and the type of output target for formatted output functions. For `printf`-style functions from module `Printf`, 'b is typically `out_channel`; for `printf`-style functions from module `Format`, 'b is typically `Format.formatter`; for `scanf`-style functions from module `Scanf`, 'b is typically `Scanf.Scanning.in_channel`.

Type argument 'b is also the type of the first argument given to user's defined printing functions for %a and %t conversions, and user's defined reading functions for %r conversion.

- 'c is the type of the result of the %a and %t printing functions, and also the type of the argument transmitted to the first argument of `kprintf`-style functions or to the `kscanf`-style functions.
- 'd is the type of parameters for the `scanf`-style functions.
- 'e is the type of the receiver function for the `scanf`-style functions.
- 'f is the final result type of a formatted input/output function invocation: for the `printf`-style functions, it is typically `unit`; for the `scanf`-style functions, it is typically the result type of the receiver function.

```

type ('a, 'b, 'c, 'd, 'e, 'f) format6 = ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.format6
type ('a, 'b, 'c, 'd) format4 = ('a, 'b, 'c, 'c, 'c, 'd) format6
type ('a, 'b, 'c) format = ('a, 'b, 'c, 'c) format4
val string_of_format : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> string
    Converts a format string into a string.

```

```

val format_of_string :
    ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
    ('a, 'b, 'c, 'd, 'e, 'f) format6
    format_of_string s returns a format string read from the string literal s. Note:
    format_of_string can not convert a string argument that is not a literal. If you need this
    functionality, use the more general Scanf.format_from_string[4] function.

```

```

val (^^) :
    ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
    ('f, 'b, 'c, 'e, 'g, 'h) format6 ->
    ('a, 'b, 'c, 'd, 'g, 'h) format6
    f1 ^^ f2 catenates format strings f1 and f2. The result is a format string that behaves as
    the concatenation of format strings f1 and f2: in case of formatted output, it accepts
    arguments from f1, then arguments from f2; in case of formatted input, it returns results
    from f1, then results from f2.

```

Program termination

```

val exit : int -> 'a
    Terminate the process, returning the given status code to the operating system: usually 0 to
    indicate no errors, and a small positive integer to indicate failure. All open output channels
    are flushed with flush_all. An implicit exit 0 is performed each time a program
    terminates normally. An implicit exit 2 is performed if the program terminates early
    because of an uncaught exception.

```

```

val at_exit : (unit -> unit) -> unit
    Register the given function to be called at program termination time. The functions
    registered with at_exit will be called when the program executes Pervasives.exit[17], or
    terminates, either normally or because of an uncaught exception. The functions are called in
    'last in, first out' order: the function most recently added with at_exit is called first.

```

18 Module Queue : First-in first-out queues.

This module implements queues (FIFOs), with in-place modification.

Warning This module is not thread-safe: each Queue.t[18] value must be protected from concurrent access (e.g. with a Mutex.t). Failure to do so can lead to a crash.

```

type 'a t

```

The type of queues containing elements of type 'a.

exception Empty

Raised when `Queue.take[18]` or `Queue.peek[18]` is applied to an empty queue.

val create : unit -> 'a t

Return a new queue, initially empty.

val add : 'a -> 'a t -> unit

add x q adds the element x at the end of the queue q.

val push : 'a -> 'a t -> unit

push is a synonym for add.

val take : 'a t -> 'a

take q removes and returns the first element in queue q, or raises **Empty** if the queue is empty.

val pop : 'a t -> 'a

pop is a synonym for take.

val peek : 'a t -> 'a

peek q returns the first element in queue q, without removing it from the queue, or raises **Empty** if the queue is empty.

val top : 'a t -> 'a

top is a synonym for peek.

val clear : 'a t -> unit

Discard all elements from a queue.

val copy : 'a t -> 'a t

Return a copy of the given queue.

val is_empty : 'a t -> bool

Return **true** if the given queue is empty, **false** otherwise.

val length : 'a t -> int

Return the number of elements in a queue.

val iter : ('a -> unit) -> 'a t -> unit

iter f q applies f in turn to all elements of q, from the least recently entered to the most recently entered. The queue itself is unchanged.

val fold : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b

`fold f accu q` is equivalent to `List.fold_left f accu l`, where `l` is the list of `q`'s elements. The queue remains unchanged.

```
val transfer : 'a t -> 'a t -> unit
```

`transfer q1 q2` adds all of `q1`'s elements at the end of the queue `q2`, then clears `q1`. It is equivalent to the sequence `iter (fun x -> add x q2) q1; clear q1`, but runs in constant time.

19 Module Genlex : A generic lexical analyzer.

This module implements a simple 'standard' lexical analyzer, presented as a function from character streams to token streams. It implements roughly the lexical conventions of OCaml, but is parameterized by the set of keywords of your language.

Example: a lexer suitable for a desk calculator is obtained by

```
let lexer = make_lexer ["+"; "-"; "*"; "/"; "let"; "="; "("; ")"]
```

The associated parser would be a function from `token stream` to, for instance, `int`, and would have rules such as:

```
let rec parse_expr = parser
  | [< n1 = parse_atom; n2 = parse_remainder n1 >] -> n2
and parse_atom = parser
  | [< 'Int n >] -> n
  | [< 'Kwd "("; n = parse_expr; 'Kwd ")" >] -> n
and parse_remainder n1 = parser
  | [< 'Kwd "+"; n2 = parse_expr >] -> n1+n2
  | [< >] -> n1
```

One should notice that the use of the `parser` keyword and associated notation for streams are only available through `camlp4` extensions. This means that one has to preprocess its sources *e. g.* by using the `"-pp"` command-line switch of the compilers.

```
type token =
  | Kwd of string
  | Ident of string
  | Int of int
  | Float of float
  | String of string
  | Char of char
```

The type of tokens. The lexical classes are: `Int` and `Float` for integer and floating-point numbers; `String` for string literals, enclosed in double quotes; `Char` for character literals, enclosed in single quotes; `Ident` for identifiers (either sequences of letters, digits, underscores and quotes, or sequences of 'operator characters' such as `+`, `*`, etc); and `Kwd` for keywords (either identifiers or single 'special characters' such as `(`, `)`, etc).

```
val make_lexer : string list -> char Stream.t -> token Stream.t
```

Construct the lexer function. The first argument is the list of keywords. An identifier `s` is returned as `Kwd s` if `s` belongs to this list, and as `Ident s` otherwise. A special character `s` is returned as `Kwd s` if `s` belongs to this list, and cause a lexical error (exception `Stream.Error` with the offending lexeme as its parameter) otherwise. Blanks and newlines are skipped. Comments delimited by `(*` and `*)` are skipped as well, and can be nested. A `Stream.Failure` exception is raised if end of stream is unexpectedly reached.

20 Module CamlinternalFormat

```
val is_in_char_set : CamlinternalFormatBasics.char_set -> char -> bool
```

```
val rev_char_set :
```

```
  CamlinternalFormatBasics.char_set -> CamlinternalFormatBasics.char_set
```

```
type mutable_char_set = bytes
```

```
val create_char_set : unit -> mutable_char_set
```

```
val add_in_char_set : mutable_char_set -> char -> unit
```

```
val freeze_char_set : mutable_char_set -> CamlinternalFormatBasics.char_set
```

```
type ('a, 'b, 'c, 'd, 'e, 'f) param_format_ebb =
```

```
  | Param_format_EBB : ('x -> 'a0, 'b0, 'c0, 'd0, 'e0, 'f0) CamlinternalFormatBasics.fmt -> (
```

```
val param_format_of_ignored_format :
```

```
  ('a, 'b, 'c, 'd, 'y, 'x) CamlinternalFormatBasics.ignored ->
```

```
  ('x, 'b, 'c, 'y, 'e, 'f) CamlinternalFormatBasics.fmt ->
```

```
  ('a, 'b, 'c, 'd, 'e, 'f) param_format_ebb
```

```
type ('b, 'c) acc_formatting_gen =
```

```
  | Acc_open_tag of ('b, 'c) acc
```

```
  | Acc_open_box of ('b, 'c) acc
```

```
type ('b, 'c) acc =
```

```
  | Acc_formatting_lit of ('b, 'c) acc * CamlinternalFormatBasics.formatting_lit
```

```
  | Acc_formatting_gen of ('b, 'c) acc
```

```
    * ('b, 'c) acc_formatting_gen
```

```
  | Acc_string_literal of ('b, 'c) acc * string
```

```
  | Acc_char_literal of ('b, 'c) acc * char
```

```
  | Acc_data_string of ('b, 'c) acc * string
```

```
  | Acc_data_char of ('b, 'c) acc * char
```

```
  | Acc_delay of ('b, 'c) acc * ('b -> 'c)
```

```
  | Acc_flush of ('b, 'c) acc
```

```
  | Acc_invalid_arg of ('b, 'c) acc * string
```

```
  | End_of_acc
```

```
type ('a, 'b) heter_list =
```

```
  | Cons : 'c * ('a0, 'b0) heter_list -> ('c -> 'a0, 'b0) heter_list
```

```
  | Nil : ('b1, 'b1) heter_list
```

```

type ('b, 'c, 'e, 'f) fmt_ebb =
  | Fmt_EBB : ('a, 'b0, 'c0, 'd, 'e0, 'f0) CamlinternalFormatBasics.fmt -> ('b0, 'c0, 'e0, 'f0)
val make_printf :
  ('b -> ('b, 'c) acc -> 'd) ->
  'b ->
  ('b, 'c) acc ->
  ('a, 'b, 'c, 'c, 'c, 'd) CamlinternalFormatBasics.fmt -> 'a
val output_acc :
  Pervasives.out_channel ->
  (Pervasives.out_channel, unit) acc -> unit
val bufput_acc : Buffer.t -> (Buffer.t, unit) acc -> unit
val strput_acc : Buffer.t -> (unit, string) acc -> unit
val type_format :
  ('x, 'b, 'c, 't, 'u, 'v) CamlinternalFormatBasics.fmt ->
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.fmtty ->
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.fmt
val fmt_ebb_of_string :
  ?legacy_behavior:bool ->
  string -> ('b, 'c, 'e, 'f) fmt_ebb
val format_of_string_fmtty :
  string ->
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.fmtty ->
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.format6
val format_of_string_format :
  string ->
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.format6 ->
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.format6
val char_of_ichonv : CamlinternalFormatBasics.int_conv -> char
val string_of_formatting_lit :
  CamlinternalFormatBasics.formatting_lit -> string
val string_of_formatting_gen :
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.formatting_gen -> string
val string_of_fmtty :
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.fmtty -> string
val string_of_fmt :
  ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.fmt -> string
val open_box_of_string : string -> int * CamlinternalFormatBasics.block_type
val symm :
  ('a1, 'b1, 'c1, 'd1, 'e1, 'f1, 'a2, 'b2, 'c2, 'd2, 'e2, 'f2)
  CamlinternalFormatBasics.fmtty_rel ->
  ('a2, 'b2, 'c2, 'd2, 'e2, 'f2, 'a1, 'b1, 'c1, 'd1, 'e1, 'f1)
  CamlinternalFormatBasics.fmtty_rel

```

```

val trans :
  ('a1, 'b1, 'c1, 'd1, 'e1, 'f1, 'a2, 'b2, 'c2, 'd2, 'e2, 'f2)
  CamlinternalFormatBasics.fmttty_rel ->
  ('a2, 'b2, 'c2, 'd2, 'e2, 'f2, 'a3, 'b3, 'c3, 'd3, 'e3, 'f3)
  CamlinternalFormatBasics.fmttty_rel ->
  ('a1, 'b1, 'c1, 'd1, 'e1, 'f1, 'a3, 'b3, 'c3, 'd3, 'e3, 'f3)
  CamlinternalFormatBasics.fmttty_rel

val recast :
  ('a1, 'b1, 'c1, 'd1, 'e1, 'f1) CamlinternalFormatBasics.fmt ->
  ('a1, 'b1, 'c1, 'd1, 'e1, 'f1, 'a2, 'b2, 'c2, 'd2, 'e2, 'f2)
  CamlinternalFormatBasics.fmttty_rel ->
  ('a2, 'b2, 'c2, 'd2, 'e2, 'f2) CamlinternalFormatBasics.fmt

```

21 Module Marshal : Marshaling of data structures.

This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of OCaml.

Warning: marshaling is currently not type-safe. The type of marshaled data is not transmitted along the value of the data, making it impossible to check that the data read back possesses the type expected by the context. In particular, the result type of the `Marshal.from_*` functions is given as `'a`, but this is misleading: the returned OCaml value does not possess type `'a` for all `'a`; it has one, unique type which cannot be determined at compile-time. The programmer should explicitly give the expected type of the returned value, using the following syntax:

- (`Marshal.from_channel chan : type`). Anything can happen at run-time if the object in the file does not belong to the given type.

Values of extensible variant types, for example exceptions (of extensible type `exn`), returned by the unmarshaller should not be pattern-matched over through `match ... with` or `try ... with`, because unmarshalling does not preserve the information required for matching their constructors. Structural equalities with other extensible variant values does not work either. Most other uses such as `Printexc.to_string`, will still work as expected.

The representation of marshaled values is not human-readable, and uses bytes that are not printable characters. Therefore, input and output channels used in conjunction with `Marshal.to_channel` and `Marshal.from_channel` must be opened in binary mode, using e.g. `open_out_bin` or `open_in_bin`; channels opened in text mode will cause unmarshaling errors on platforms where text channels behave differently than binary channels, e.g. Windows.

```

type extern_flags =
  | No_sharing
      Don't preserve sharing
  | Closures
      Send function closures

```

| Compat_32

Ensure 32-bit compatibility

The flags to the `Marshal.to_*` functions below.

```
val to_channel : Pervasives.out_channel -> 'a -> extern_flags list -> unit
```

`Marshal.to_channel chan v flags` writes the representation of `v` on channel `chan`. The `flags` argument is a possibly empty list of flags that governs the marshaling behavior with respect to sharing, functional values, and compatibility between 32- and 64-bit platforms.

If `flags` does not contain `Marshal.No_sharing`, circularities and sharing inside the value `v` are detected and preserved in the sequence of bytes produced. In particular, this guarantees that marshaling always terminates. Sharing between values marshaled by successive calls to `Marshal.to_channel` is neither detected nor preserved, though. If `flags` contains `Marshal.No_sharing`, sharing is ignored. This results in faster marshaling if `v` contains no shared substructures, but may cause slower marshaling and larger byte representations if `v` actually contains sharing, or even non-termination if `v` contains cycles.

If `flags` does not contain `Marshal.Closures`, marshaling fails when it encounters a functional value inside `v`: only 'pure' data structures, containing neither functions nor objects, can safely be transmitted between different programs. If `flags` contains `Marshal.Closures`, functional values will be marshaled as a the position in the code of the program together with the values corresponding to the free variables captured in the closure. In this case, the output of marshaling can only be read back in processes that run exactly the same program, with exactly the same compiled code. (This is checked at un-marshaling time, using an MD5 digest of the code transmitted along with the code position.)

The exact definition of which free variables are captured in a closure is not specified and can vary between bytecode and native code (and according to optimization flags). In particular, a function value accessing a global reference may or may not include the reference in its closure. If it does, unmarshaling the corresponding closure will create a new reference, different from the global one.

If `flags` contains `Marshal.Compat_32`, marshaling fails when it encounters an integer value outside the range $[-2^{30}, 2^{30}-1]$ of integers that are representable on a 32-bit platform. This ensures that marshaled data generated on a 64-bit platform can be safely read back on a 32-bit platform. If `flags` does not contain `Marshal.Compat_32`, integer values outside the range $[-2^{30}, 2^{30}-1]$ are marshaled, and can be read back on a 64-bit platform, but will cause an error at un-marshaling time when read back on a 32-bit platform. The `Marshal.Compat_32` flag only matters when marshaling is performed on a 64-bit platform; it has no effect if marshaling is performed on a 32-bit platform.

```
val to_bytes : 'a -> extern_flags list -> bytes
```

`Marshal.to_bytes v flags` returns a byte sequence containing the representation of `v`. The `flags` argument has the same meaning as for `Marshal.to_channel`[21].

Since: 4.02.0

```
val to_string : 'a -> extern_flags list -> string
```

Same as `to_bytes` but return the result as a string instead of a byte sequence.

`val to_buffer : bytes -> int -> int -> 'a -> extern_flags list -> int`

`Marshal.to_buffer buff ofs len v flags` marshals the value `v`, storing its byte representation in the sequence `buff`, starting at index `ofs`, and writing at most `len` bytes. It returns the number of bytes actually written to the sequence. If the byte representation of `v` does not fit in `len` characters, the exception `Failure` is raised.

`val from_channel : Pervasives.in_channel -> 'a`

`Marshal.from_channel chan` reads from channel `chan` the byte representation of a structured value, as produced by one of the `Marshal.to_*` functions, and reconstructs and returns the corresponding value.

`val from_bytes : bytes -> int -> 'a`

`Marshal.from_bytes buff ofs` unmarshals a structured value like `Marshal.from_channel[21]` does, except that the byte representation is not read from a channel, but taken from the byte sequence `buff`, starting at position `ofs`. The byte sequence is not mutated.

Since: 4.02.0

`val from_string : string -> int -> 'a`

Same as `from_bytes` but take a string as argument instead of a byte sequence.

`val header_size : int`

The bytes representing a marshaled value are composed of a fixed-size header and a variable-sized data part, whose size can be determined from the header.

`Marshal.header_size[21]` is the size, in bytes, of the header. `Marshal.data_size[21] buff ofs` is the size, in bytes, of the data part, assuming a valid header is stored in `buff` starting at position `ofs`. Finally, `Marshal.total_size[21] buff ofs` is the total size, in bytes, of the marshaled value. Both `Marshal.data_size[21]` and `Marshal.total_size[21]` raise `Failure` if `buff, ofs` does not contain a valid header.

To read the byte representation of a marshaled value into a byte sequence, the program needs to read first `Marshal.header_size[21]` bytes into the sequence, then determine the length of the remainder of the representation using `Marshal.data_size[21]`, make sure the sequence is large enough to hold the remaining data, then read it, and finally call `Marshal.from_bytes[21]` to unmarshal the value.

`val data_size : bytes -> int -> int`

See `Marshal.header_size[21]`.

`val total_size : bytes -> int -> int`

See `Marshal.header_size[21]`.

22 Module Sys : System interface.

Every function in this module raises **Sys_error** with an informative message when the underlying system call signal an error.

val argv : string array

The command line arguments given to the process. The first element is the command name used to invoke the program. The following elements are the command-line arguments given to the program.

val executable_name : string

The name of the file containing the executable currently running.

val file_exists : string -> bool

Test if a file with the given name exists.

val is_directory : string -> bool

Returns **true** if the given name refers to a directory, **false** if it refers to another kind of file. Raise **Sys_error** if no file exists with the given name.

Since: 3.10.0

val remove : string -> unit

Remove the given file name from the file system.

val rename : string -> string -> unit

Rename a file. The first argument is the old name and the second is the new name. If there is already another file under the new name, **rename** may replace it, or raise an exception, depending on your operating system.

val getenv : string -> string

Return the value associated to a variable in the process environment. Raise **Not_found** if the variable is unbound.

val command : string -> int

Execute the given shell command and return its exit code.

val time : unit -> float

Return the processor time, in seconds, used by the program since the beginning of execution.

val chdir : string -> unit

Change the current working directory of the process.

val getcwd : unit -> string

Return the current working directory of the process.

val readdir : string -> string array

Return the names of all files present in the given directory. Names denoting the current directory and the parent directory ("." and ".." in Unix) are not returned. Each string in the result is a file name rather than a complete path. There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

val interactive : bool Pervasives.ref

This reference is initially set to **false** in standalone programs and to **true** if the code is being executed under the interactive toplevel system **ocaml**.

val os_type : string

Operating system currently executing the OCaml program. One of

- "Unix" (for all Unix versions, including Linux and Mac OS X),
- "Win32" (for MS-Windows, OCaml compiled with MSVC++ or Mingw),
- "Cygwin" (for MS-Windows, OCaml compiled with Cygwin).

val unix : bool

True if **Sys.os_type** = "Unix".

Since: 4.01.0

val win32 : bool

True if **Sys.os_type** = "Win32".

Since: 4.01.0

val cygwin : bool

True if **Sys.os_type** = "Cygwin".

Since: 4.01.0

val word_size : int

Size of one word on the machine currently executing the OCaml program, in bits: 32 or 64.

val big_endian : bool

Whether the machine currently executing the Caml program is big-endian.

Since: 4.00.0

val max_string_length : int

Maximum length of strings and byte sequences.

val max_array_length : int

Maximum length of a normal array. The maximum length of a float array is **max_array_length/2** on 32-bit machines and **max_array_length** on 64-bit machines.

Signal handling

```
type signal_behavior =  
  | Signal_default  
  | Signal_ignore  
  | Signal_handle of (int -> unit)
```

What to do when receiving a signal:

- **Signal_default**: take the default behavior (usually: abort the program)
- **Signal_ignore**: ignore the signal
- **Signal_handle f**: call function **f**, giving it the signal number as argument.

```
val signal : int -> signal_behavior -> signal_behavior
```

Set the behavior of the system on receipt of a given signal. The first argument is the signal number. Return the behavior previously associated with the signal. If the signal number is invalid (or not available on your system), an **Invalid_argument** exception is raised.

```
val set_signal : int -> signal_behavior -> unit
```

Same as **Sys.signal[22]** but return value is ignored.

Signal numbers for the standard POSIX signals.

```
val sigabrt : int
```

Abnormal termination

```
val sigalrm : int
```

Timeout

```
val sigfpe : int
```

Arithmetic exception

```
val sighup : int
```

Hangup on controlling terminal

```
val sigill : int
```

Invalid hardware instruction

```
val sigint : int
```

Interactive interrupt (ctrl-C)

```
val sigkill : int
```

Termination (cannot be ignored)

```
val sigpipe : int
```

Broken pipe

```

val sigquit : int
    Interactive termination

val sigsegv : int
    Invalid memory reference

val sigterm : int
    Termination

val sigusr1 : int
    Application-defined signal 1

val sigusr2 : int
    Application-defined signal 2

val sigchld : int
    Child process terminated

val sigcont : int
    Continue

val sigstop : int
    Stop

val sigtstp : int
    Interactive stop

val sigttin : int
    Terminal read from background process

val sigttou : int
    Terminal write from background process

val sigvtalrm : int
    Timeout in virtual time

val sigprof : int
    Profiling interrupt

exception Break
    Exception raised on interactive interrupt if Sys.catch_break[22] is on.

val catch_break : bool -> unit

```

`catch_break` governs whether interactive interrupt (ctrl-C) terminates the program or raises the `Break` exception. Call `catch_break true` to enable raising `Break`, and `catch_break false` to let the system terminate the program on user interrupt.

`val ocaml_version : string`

`ocaml_version` is the version of OCaml. It is a string of the form "major.minor[.patchlevel][+additional-info]", where `major`, `minor`, and `patchlevel` are integers, and `additional-info` is an arbitrary string. The `[.patchlevel]` and `[+additional-info]` parts may be absent.

23 Module `StringLabels` : String operations.

`val length : string -> int`

Return the length (number of characters) of the given string.

`val get : string -> int -> char`

`String.get s n` returns the character at index `n` in string `s`. You can also write `s.[n]` instead of `String.get s n`.

Raise `Invalid_argument` if `n` not a valid index in `s`.

`val set : bytes -> int -> char -> unit`

Deprecated. This is a deprecated alias of `BytesLabels.set[14].String.set s n c` modifies byte sequence `s` in place, replacing the byte at index `n` with `c`. You can also write `s.[n] <- c` instead of `String.set s n c`.

Raise `Invalid_argument` if `n` is not a valid index in `s`.

`val create : int -> bytes`

Deprecated. This is a deprecated alias of `BytesLabels.create[14].String.create n` returns a fresh byte sequence of length `n`. The sequence is uninitialized and contains arbitrary bytes.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val make : int -> char -> string`

`String.make n c` returns a fresh string of length `n`, filled with the character `c`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val init : int -> f:(int -> char) -> string`

`init n f` returns a string of length `n`, with character `i` initialized to the result of `f i`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

`val copy : string -> string`

Return a copy of the given string.

```

val sub : string -> pos:int -> len:int -> string
    String.sub s start len returns a fresh string of length len, containing the substring of s
    that starts at position start and has length len.
    Raise Invalid_argument if start and len do not designate a valid substring of s.

val fill : bytes -> pos:int -> len:int -> char -> unit
    Deprecated. This is a deprecated alias of BytesLabels.fill[14].String.fill s start len
    c modifies byte sequence s in place, replacing len bytes by c, starting at start.
    Raise Invalid_argument if start and len do not designate a valid substring of s.

val blit :
    src:string -> src_pos:int -> dst:bytes -> dst_pos:int -> len:int -> unit
    String.blit src srcoff dst dstoff len copies len bytes from the string src, starting at
    index srcoff, to byte sequence dst, starting at character number dstoff.
    Raise Invalid_argument if srcoff and len do not designate a valid range of src, or if
    dstoff and len do not designate a valid range of dst.

val concat : sep:string -> string list -> string
    String.concat sep sl concatenates the list of strings sl, inserting the separator string sep
    between each.

val iter : f:(char -> unit) -> string -> unit
    String.iter f s applies function f in turn to all the characters of s. It is equivalent to f
    s.[0]; f s.[1]; ...; f s.[String.length s - 1]; ().

val iteri : f:(int -> char -> unit) -> string -> unit
    Same as String.iter[44], but the function is applied to the index of the element as first
    argument (counting from 0), and the character itself as second argument.
    Since: 4.00.0

val map : f:(char -> char) -> string -> string
    String.map f s applies function f in turn to all the characters of s and stores the results in
    a new string that is returned.
    Since: 4.00.0

val mapi : f:(int -> char -> char) -> string -> string
    String.mapi f s calls f with each character of s and its index (in increasing index order)
    and stores the results in a new string that is returned.
    Since: 4.02.0

val trim : string -> string

```

Return a copy of the argument, without leading and trailing whitespace. The characters regarded as whitespace are: ' ', '\012', '\n', '\r', and '\t'. If there is no leading nor trailing whitespace character in the argument, return the original string itself, not a copy.

Since: 4.00.0

val escaped : string -> string

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of OCaml. If there is no special character in the argument, return the original string itself, not a copy. Its inverse function is `Scanf.unescaped`.

val index : string -> char -> int

`String.index s c` returns the index of the first occurrence of character `c` in string `s`.

Raise `Not_found` if `c` does not occur in `s`.

val rindex : string -> char -> int

`String.rindex s c` returns the index of the last occurrence of character `c` in string `s`.

Raise `Not_found` if `c` does not occur in `s`.

val index_from : string -> int -> char -> int

`String.index_from s i c` returns the index of the first occurrence of character `c` in string `s` after position `i`. `String.index s c` is equivalent to `String.index_from s 0 c`.

Raise `Invalid_argument` if `i` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` after position `i`.

val rindex_from : string -> int -> char -> int

`String.rindex_from s i c` returns the index of the last occurrence of character `c` in string `s` before position `i+1`. `String.rindex s c` is equivalent to `String.rindex_from s (String.length s - 1) c`.

Raise `Invalid_argument` if `i+1` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` before position `i+1`.

val contains : string -> char -> bool

`String.contains s c` tests if character `c` appears in the string `s`.

val contains_from : string -> int -> char -> bool

`String.contains_from s start c` tests if character `c` appears in `s` after position `start`.

`String.contains s c` is equivalent to `String.contains_from s 0 c`.

Raise `Invalid_argument` if `start` is not a valid position in `s`.

val rcontains_from : string -> int -> char -> bool

`String.rcontains_from s stop c` tests if character `c` appears in `s` before position `stop+1`.

Raise `Invalid_argument` if `stop < 0` or `stop+1` is not a valid position in `s`.

```
val uppercase : string -> string
    Return a copy of the argument, with all lowercase letters translated to uppercase, including
    accented letters of the ISO Latin-1 (8859-1) character set.
```

```
val lowercase : string -> string
    Return a copy of the argument, with all uppercase letters translated to lowercase, including
    accented letters of the ISO Latin-1 (8859-1) character set.
```

```
val capitalize : string -> string
    Return a copy of the argument, with the first character set to uppercase.
```

```
val uncapitalize : string -> string
    Return a copy of the argument, with the first character set to lowercase.
```

```
type t = string
    An alias for the type of strings.
```

```
val compare : t -> t -> int
    The comparison function for strings, with the same specification as
    Pervasives.compare[17]. Along with the type t, this function compare allows the module
    String to be passed as argument to the functors Set.Make[38] and Map.Make[3].
```

24 Module Printf : Formatted output functions.

```
val fprintf :
    Pervasives.out_channel ->
    ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
    fprintf outchan format arg1 ... argN formats the arguments arg1 to argN according
    to the format string format, and outputs the resulting string on the channel outchan.
```

The format string is a character string which contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which causes conversion and printing of arguments.

Conversion specifications have the following form:

```
% [flags] [width] [.precision] type
```

In short, a conversion specification consists in the `%` character, followed by optional modifiers and a type which is made of one or two characters.

The types and their meanings are:

- `d`, `i`: convert an integer argument to signed decimal.
- `u`, `n`, `l`, `L`, or `N`: convert an integer argument to unsigned decimal. Warning: `n`, `l`, `L`, and `N` are used for `scanf`, and should not be used for `printf`.

- **x**: convert an integer argument to unsigned hexadecimal, using lowercase letters.
- **X**: convert an integer argument to unsigned hexadecimal, using uppercase letters.
- **o**: convert an integer argument to unsigned octal.
- **s**: insert a string argument.
- **S**: convert a string argument to OCaml syntax (double quotes, escapes).
- **c**: insert a character argument.
- **C**: convert a character argument to OCaml syntax (single quotes, escapes).
- **f**: convert a floating-point argument to decimal notation, in the style `dddd.ddd`.
- **F**: convert a floating-point argument to OCaml syntax (`dddd.` or `dddd.ddd` or `d.ddd e+-dd`).
- **e** or **E**: convert a floating-point argument to decimal notation, in the style `d.ddd e+-dd` (mantissa and exponent).
- **g** or **G**: convert a floating-point argument to decimal notation, in style **f** or **e**, **E** (whichever is more compact).
- **B**: convert a boolean argument to the string `true` or `false`
- **b**: convert a boolean argument (deprecated; do not use in new programs).
- **ld**, **li**, **lu**, **lx**, **lX**, **lo**: convert an `int32` argument to the format specified by the second letter (decimal, hexadecimal, etc).
- **nd**, **ni**, **nu**, **nx**, **nX**, **no**: convert a `nativeint` argument to the format specified by the second letter.
- **Ld**, **Li**, **Lu**, **Lx**, **LX**, **Lo**: convert an `int64` argument to the format specified by the second letter.
- **a**: user-defined printer. Take two arguments and apply the first one to `outchan` (the current output channel) and to the second argument. The first argument must therefore have type `out_channel -> 'b -> unit` and the second `'b`. The output produced by the function is inserted in the output of `fprintf` at the current point.
- **t**: same as `%a`, but take only one argument (with type `out_channel -> unit`) and apply it to `outchan`.
- **{ fmt %}**: convert a format string argument to its type digest. The argument must have the same type as the internal format string `fmt`.
- **(fmt %)**: format string substitution. Take a format string argument and substitute it to the internal format string `fmt` to print following arguments. The argument must have the same type as the internal format string `fmt`.
- **!**: take no argument and flush the output.
- **%**: take no argument and output one `%` character.
- **@**: take no argument and output one `@` character.
- **,**: take no argument and output nothing: a no-op delimiter for conversion specifications.

The optional **flags** are:

- -: left-justify the output (default is right justification).
- 0: for numerical conversions, pad with zeroes instead of spaces.
- +: for signed numerical conversions, prefix number with a + sign if positive.
- space: for signed numerical conversions, prefix number with a space if positive.
- #: request an alternate formatting style for the hexadecimal and octal integer types (x, X, o, lx, lX, lo, Lx, LX, Lo).

The optional **width** is an integer indicating the minimal width of the result. For instance, **%6d** prints an integer, prefixing it with spaces to fill at least 6 characters.

The optional **precision** is a dot . followed by an integer indicating how many digits follow the decimal point in the **%f**, **%e**, and **%E** conversions. For instance, **%.4f** prints a **float** with 4 fractional digits.

The integer in a **width** or **precision** can also be specified as *, in which case an extra integer argument is taken to specify the corresponding **width** or **precision**. This integer argument precedes immediately the argument to print. For instance, **%.*f** prints a **float** with as many fractional digits as the value of the argument given before the float.

```
val printf : ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
    Same as Printf.fprintf[24], but output on stdout.
```

```
val eprintf : ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
    Same as Printf.fprintf[24], but output on stderr.
```

```
val sprintf : ('a, unit, string) Pervasives.format -> 'a
    Same as Printf.fprintf[24], but instead of printing on an output channel, return a string
    containing the result of formatting the arguments.
```

```
val bprintf : Buffer.t -> ('a, Buffer.t, unit) Pervasives.format -> 'a
    Same as Printf.fprintf[24], but instead of printing on an output channel, append the
    formatted arguments to the given extensible buffer (see module Buffer[32]).
```

```
val ifprintf : 'a -> ('b, 'a, unit) Pervasives.format -> 'b
    Same as Printf.fprintf[24], but does not print anything. Useful to ignore some material
    when conditionally printing.
```

Since: 3.10.0

Formatted output functions with continuations.

```
val kfprintf :
  (Pervasives.out_channel -> 'a) ->
  Pervasives.out_channel ->
  ('b, Pervasives.out_channel, unit, 'a) Pervasives.format4 -> 'b
```

Same as **fprintf**, but instead of returning immediately, passes the out channel to its first argument at the end of printing.

Since: 3.09.0

```

val ikfprintf :
  (Pervasives.out_channel -> 'a) ->
  Pervasives.out_channel ->
  ('b, Pervasives.out_channel, unit, 'a) Pervasives.format4 -> 'b
  Same as kfprintf above, but does not print anything. Useful to ignore some material when
  conditionally printing.
  Since: 4.0

val ksprintf :
  (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b
  Same as sprintf above, but instead of returning the string, passes it to the first argument.
  Since: 3.09.0

val kbprintf :
  (Buffer.t -> 'a) ->
  Buffer.t -> ('b, Buffer.t, unit, 'a) Pervasives.format4 -> 'b
  Same as bprintf, but instead of returning immediately, passes the buffer to its first
  argument at the end of printing.
  Since: 3.10.0

  Deprecated
val kprintf :
  (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b
  A deprecated synonym for ksprintf.

```

25 Module CamlinternalLazy : Run-time support for lazy values.

All functions in this module are for system use only, not for the casual user.

```

exception Undefined
val force_lazy_block : 'a lazy_t -> 'a
val force_val_lazy_block : 'a lazy_t -> 'a
val force : 'a lazy_t -> 'a
val force_val : 'a lazy_t -> 'a

```

26 Module ListLabels : 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. Raise **Invalid_argument** "List.nth" if *n* is negative.

```
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
```

`ListLabels.rev_append l1 l2` reverses *l1* and concatenates it to *l2*. This is equivalent to `ListLabels.rev[26] 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
```

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

Iterators

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

`ListLabels.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 : f:(int -> 'a -> unit) -> 'a list -> unit
```

Same as `ListLabels.iter[26]`, 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 : f:('a -> 'b) -> 'a list -> 'b list
    ListLabels.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 : f:(int -> 'a -> 'b) -> 'a list -> 'b list
    Same as ListLabels.map[26], 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 rev_map : f:('a -> 'b) -> 'a list -> 'b list
    ListLabels.rev_map f l gives the same result as ListLabels.rev[26]
    (ListLabels.map[26] f l), but is tail-recursive and more efficient.

val fold_left : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a
    ListLabels.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
    ListLabels.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
    ListLabels.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
    ListLabels.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
    ListLabels.rev_map2 f l1 l2 gives the same result as ListLabels.rev[26]
    (ListLabels.map2[26] f l1 l2), but is tail-recursive and more efficient.

val fold_left2 :
    f:('a -> 'b -> 'c -> 'a) -> init:'a -> 'b list -> 'c list -> 'a
    ListLabels.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
    ListLabels.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
  Same as ListLabels.for_all[26], but for a two-argument predicate. Raise
  Invalid_argument if the two lists have different lengths.

val exists2 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
  Same as ListLabels.exists[26], but for a two-argument predicate. Raise
  Invalid_argument if the two lists have different lengths.

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

val memq : 'a -> set:'a list -> bool
  Same as ListLabels.mem[26], 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
  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 : f:('a -> bool) -> 'a list -> 'a list
  find_all is another name for ListLabels.filter[26].

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 `ListLabels.assoc[26]`, but uses physical equality instead of structural equality to compare keys.

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

Same as `ListLabels.assoc[26]`, but simply return true if a binding exists, and false if no bindings exist for the given key.

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

Same as `ListLabels.mem_assoc[26]`, 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 `ListLabels.remove_assoc[26]`, 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 : cmp:('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, `Pervasives.compare[17]` is a suitable comparison function. The resulting list is sorted in increasing order. `ListLabels.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 : cmp:('a -> 'a -> int) -> 'a list -> 'a list
```

Same as `ListLabels.sort[26]`, 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 : cmp:('a -> 'a -> int) -> 'a list -> 'a list
```

Same as `ListLabels.sort[26]` or `ListLabels.stable_sort[26]`, whichever is faster on typical input.

```
val merge : cmp:('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).

27 Module `Hashtbl` : Hash tables and hash functions.

Hash tables are hashed association tables, with in-place modification.

Generic interface

```
type ('a, 'b) t
```

The type of hash tables from type `'a` to type `'b`.

```
val create : ?random:bool -> int -> ('a, 'b) t
```

`Hashtbl.create n` creates a new, empty hash table, with initial size `n`. For best results, `n` should be on the order of the expected number of elements that will be in the table. The table grows as needed, so `n` is just an initial guess.

The optional `random` parameter (a boolean) controls whether the internal organization of the hash table is randomized at each execution of `Hashtbl.create` or deterministic over all executions.

A hash table that is created with `~random:false` uses a fixed hash function (`Hashtbl.hash[27]`) to distribute keys among buckets. As a consequence, collisions between keys happen deterministically. In Web-facing applications or other security-sensitive applications, the deterministic collision patterns can be exploited by a malicious user to create a denial-of-service attack: the attacker sends input crafted to create many collisions in the table, slowing the application down.

A hash table that is created with `~random:true` uses the seeded hash function `Hashtbl.seeded_hash[27]` with a seed that is randomly chosen at hash table creation time. In effect, the hash function used is randomly selected among $2^{\{30\}}$ different hash functions. All these hash functions have different collision patterns, rendering ineffective the denial-of-service attack described above. However, because of randomization, enumerating

all elements of the hash table using `Hashtbl.fold[27]` or `Hashtbl.iter[27]` is no longer deterministic: elements are enumerated in different orders at different runs of the program.

If no `~random` parameter is given, hash tables are created in non-random mode by default. This default can be changed either programmatically by calling `Hashtbl.randomize[27]` or by setting the `R` flag in the `OCAMLRUNPARAM` environment variable.

Before 4.00.0 the `random` parameter was not present and all hash tables were created in non-randomized mode.

val clear : ('a, 'b) t -> unit

Empty a hash table. Use `reset` instead of `clear` to shrink the size of the bucket table to its initial size.

val reset : ('a, 'b) t -> unit

Empty a hash table and shrink the size of the bucket table to its initial size.

Since: 4.00.0

val copy : ('a, 'b) t -> ('a, 'b) t

Return a copy of the given hashtable.

val add : ('a, 'b) t -> 'a -> 'b -> unit

`Hashtbl.add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `Hashtbl.remove[27] tbl x`, the previous binding for `x`, if any, is restored. (Same behavior as with association lists.)

val find : ('a, 'b) t -> 'a -> 'b

`Hashtbl.find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

val find_all : ('a, 'b) t -> 'a -> 'b list

`Hashtbl.find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

val mem : ('a, 'b) t -> 'a -> bool

`Hashtbl.mem tbl x` checks if `x` is bound in `tbl`.

val remove : ('a, 'b) t -> 'a -> unit

`Hashtbl.remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.

val replace : ('a, 'b) t -> 'a -> 'b -> unit

`Hashtbl.replace tbl x y` replaces the current binding of `x` in `tbl` by a binding of `x` to `y`. If `x` is unbound in `tbl`, a binding of `x` to `y` is added to `tbl`. This is functionally equivalent to `Hashtbl.remove[27] tbl x` followed by `Hashtbl.add[27] tbl x y`.

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
```

`Hashtbl.iter f tbl` applies `f` to all bindings in table `tbl`. `f` receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to `f`.

The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

If the hash table was created in non-randomized mode, the order in which the bindings are enumerated is reproducible between successive runs of the program, and even between minor versions of OCaml. For randomized hash tables, the order of enumeration is entirely random.

```
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

`Hashtbl.fold f tbl init` computes `(f kN dN ... (f k1 d1 init) ...)`, where `k1 ... kN` are the keys of all bindings in `tbl`, and `d1 ... dN` are the associated values. Each binding is presented exactly once to `f`.

The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

If the hash table was created in non-randomized mode, the order in which the bindings are enumerated is reproducible between successive runs of the program, and even between minor versions of OCaml. For randomized hash tables, the order of enumeration is entirely random.

```
val length : ('a, 'b) t -> int
```

`Hashtbl.length tbl` returns the number of bindings in `tbl`. It takes constant time. Multiple bindings are counted once each, so `Hashtbl.length` gives the number of times `Hashtbl.iter` calls its first argument.

```
val randomize : unit -> unit
```

After a call to `Hashtbl.randomize()`, hash tables are created in randomized mode by default: `Hashtbl.create[27]` returns randomized hash tables, unless the `~random:false` optional parameter is given. The same effect can be achieved by setting the `R` parameter in the `OCAMLRUNPARAM` environment variable.

It is recommended that applications or Web frameworks that need to protect themselves against the denial-of-service attack described in `Hashtbl.create[27]` call `Hashtbl.randomize()` at initialization time.

Note that once `Hashtbl.randomize()` was called, there is no way to revert to the non-randomized default behavior of `Hashtbl.create[27]`. This is intentional.

Non-randomized hash tables can still be created using `Hashtbl.create ~random:false`.

Since: 4.00.0

```
type statistics = {  
  num_bindings : int ;
```

Number of bindings present in the table. Same value as returned by `Hashtbl.length[27]`.

`num_buckets : int ;`

Number of buckets in the table.

`max_bucket_length : int ;`

Maximal number of bindings per bucket.

`bucket_histogram : int array ;`

Histogram of bucket sizes. This array `histo` has length `max_bucket_length + 1`. The value of `histo.(i)` is the number of buckets whose size is `i`.

}

`val stats : ('a, 'b) t -> statistics`

`Hashtbl.stats tbl` returns statistics about the table `tbl`: number of buckets, size of the biggest bucket, distribution of buckets by size.

Since: 4.00.0

Functorial interface

Functorial interface

The functorial interface allows the use of specific comparison and hash functions, either for performance/security concerns, or because keys are not hashable/comparable with the polymorphic builtins.

For instance, one might want to specialize a table for integer keys:

```
module IntHash =
  struct
    type t = int
    let equal i j = i=j
    let hash i = i land max_int
  end

module IntHashtbl = Hashtbl.Make(IntHash)

let h = IntHashtbl.create 17 in
IntHashtbl.add h 12 "hello";;
```

This creates a new module `IntHashtbl`, with a new type `'a IntHashtbl.t` of tables from `int` to `'a`. In this example, `h` contains `string` values so its type is `string IntHashtbl.t`.

Note that the new type `'a IntHashtbl.t` is not compatible with the type `('a, 'b) Hashtbl.t` of the generic interface. For example, `Hashtbl.length h` would not type-check, you must use `IntHashtbl.length`.

```
module type HashedType =
  sig
    type t
```

The type of the hashtable keys.

```
val equal : t -> t -> bool
```

The equality predicate used to compare keys.

```
val hash : t -> int
```

A hashing function on keys. It must be such that if two keys are equal according to `equal`, then they have identical hash values as computed by `hash`. Examples: suitable (`equal`, `hash`) pairs for arbitrary key types include

- `((=), Hashtbl.hash[27])` for comparing objects by structure (provided objects do not contain floats)
- `((fun x y -> compare x y = 0), Hashtbl.hash[27])` for comparing objects by structure and handling `Pervasives.nan[17]` correctly
- `((==), Hashtbl.hash[27])` for comparing objects by physical equality (e.g. for mutable or cyclic objects).

```
end
```

The input signature of the functor `Hashtbl.Make[27]`.

```
module type S =
sig
  type key
  type 'a t
  val create : int -> 'a t
  val clear : 'a t -> unit
  val reset : 'a t -> unit
  val copy : 'a t -> 'a t
  val add : 'a t -> key -> 'a -> unit
  val remove : 'a t -> key -> unit
  val find : 'a t -> key -> 'a
  val find_all : 'a t -> key -> 'a list
  val replace : 'a t -> key -> 'a -> unit
  val mem : 'a t -> key -> bool
  val iter : (key -> 'a -> unit) -> 'a t -> unit
  val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  val length : 'a t -> int
  val stats : 'a t -> Hashtbl.statistics
end
```

The output signature of the functor `Hashtbl.Make[27]`.

```

module Make :
  functor (H : HashedType) -> S with type key = H.t
    Functor building an implementation of the hashtable structure. The functor Hashtbl.Make
    returns a structure containing a type key of keys and a type 'a t of hash tables associating
    data of type 'a to keys of type key. The operations perform similarly to those of the generic
    interface, but use the hashing and equality functions specified in the functor argument H
    instead of generic equality and hashing. Since the hash function is not seeded, the create
    operation of the result structure always returns non-randomized hash tables.

module type SeededHashedType =
  sig
    type t
      The type of the hashtable keys.

    val equal : t -> t -> bool
      The equality predicate used to compare keys.

    val hash : int -> t -> int
      A seeded hashing function on keys. The first argument is the seed. It must be the case
      that if equal x y is true, then hash seed x = hash seed y for any value of seed. A
      suitable choice for hash is the function Hashtbl.seeded_hash[27] below.

  end

  The input signature of the functor Hashtbl.MakeSeeded[27].
  Since: 4.00.0

module type SeededS =
  sig
    type key
    type 'a t
    val create : ?random:bool -> int -> 'a t
    val clear : 'a t -> unit
    val reset : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
  end

```

```

val iter : (key -> 'a -> unit) -> 'a t -> unit
val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
val length : 'a t -> int
val stats : 'a t -> Hashtbl.statistics
end

```

The output signature of the functor `Hashtbl.MakeSeeded`[27].

Since: 4.00.0

module MakeSeeded :

```

functor (H : SeededHashedType) -> SeededS with type key = H.t

```

Functor building an implementation of the hashtable structure. The functor `Hashtbl.MakeSeeded` returns a structure containing a type `key` of keys and a type `'a t` of hash tables associating data of type `'a` to keys of type `key`. The operations perform similarly to those of the generic interface, but use the seeded hashing and equality functions specified in the functor argument `H` instead of generic equality and hashing. The `create` operation of the result structure supports the `~random` optional parameter and returns randomized hash tables if `~random:true` is passed or if randomization is globally on (see `Hashtbl.randomize`[27]).

Since: 4.00.0

The polymorphic hash functions

```

val hash : 'a -> int

```

`Hashtbl.hash x` associates a nonnegative integer to any value of any type. It is guaranteed that if `x = y` or `Pervasives.compare x y = 0`, then `hash x = hash y`. Moreover, `hash` always terminates, even on cyclic structures.

```

val seeded_hash : int -> 'a -> int

```

A variant of `Hashtbl.hash`[27] that is further parameterized by an integer seed.

Since: 4.00.0

```

val hash_param : int -> int -> 'a -> int

```

`Hashtbl.hash_param meaningful total x` computes a hash value for `x`, with the same properties as for `hash`. The two extra integer parameters `meaningful` and `total` give more precise control over hashing. Hashing performs a breadth-first, left-to-right traversal of the structure `x`, stopping after `meaningful` meaningful nodes were encountered, or `total` nodes (meaningful or not) were encountered. If `total` as specified by the user exceeds a certain value, currently 256, then it is capped to that value. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of `meaningful` and `total` means that more nodes are taken into account to compute the final hash value, and therefore collisions are less likely to happen. However, hashing takes longer. The parameters `meaningful` and `total` govern the tradeoff between accuracy and speed. As default choices, `Hashtbl.hash`[27] and `Hashtbl.seeded_hash`[27] take `meaningful = 10` and `total = 100`.

```
val seeded_hash_param : int -> int -> int -> 'a -> int
```

A variant of `Hashtbl.hash_param`[27] that is further parameterized by an integer seed.

Usage: `Hashtbl.seeded_hash_param meaningful total seed x`.

Since: 4.00.0

28 Module MoreLabels : Extra labeled libraries.

This meta-module provides labeled version of the `Hashtbl`[27], `Map`[3] and `Set`[38] modules.

They only differ by their labels. They are provided to help porting from previous versions of OCaml. The contents of this module are subject to change.

```
module Hashtbl :
sig
  type ('a, 'b) t = ('a, 'b) Hashtbl.t
  val create : ?random:bool -> int -> ('a, 'b) t
  val clear : ('a, 'b) t -> unit
  val reset : ('a, 'b) t -> unit
  val copy : ('a, 'b) t -> ('a, 'b) t
  val add : ('a, 'b) t -> key:'a -> data:'b -> unit
  val find : ('a, 'b) t -> 'a -> 'b
  val find_all : ('a, 'b) t -> 'a -> 'b list
  val mem : ('a, 'b) t -> 'a -> bool
  val remove : ('a, 'b) t -> 'a -> unit
  val replace : ('a, 'b) t -> key:'a -> data:'b -> unit
  val iter : f:(key:'a -> data:'b -> unit) -> ('a, 'b) t -> unit
  val fold : f:(key:'a -> data:'b -> 'c -> 'c) ->
    ('a, 'b) t -> init:'c -> 'c
  val length : ('a, 'b) t -> int
  val randomize : unit -> unit
  type statistics = Hashtbl.statistics
  val stats : ('a, 'b) t -> statistics
  module type HashedType =
    Hashtbl.HashedType
  module type SeededHashedType =
    Hashtbl.SeededHashedType
  module type S =
    sig
```

```

type key
type 'a t
val create : int -> 'a t
val clear : 'a t -> unit
val reset : 'a t -> unit
val copy : 'a t -> 'a t
val add : 'a t -> key:key -> data:'a -> unit
val remove : 'a t -> key -> unit
val find : 'a t -> key -> 'a
val find_all : 'a t -> key -> 'a list
val replace : 'a t -> key:key -> data:'a -> unit
val mem : 'a t -> key -> bool
val iter : f:(key:key -> data:'a -> unit) ->
  'a t -> unit
val fold : f:(key:key -> data:'a -> 'b -> 'b) ->
  'a t -> init:'b -> 'b
val length : 'a t -> int
val stats : 'a t -> MoreLabels.Hashtbl.statistics
end

module type SeededS =
sig
  type key
  type 'a t
  val create : ?random:bool -> int -> 'a t
  val clear : 'a t -> unit
  val reset : 'a t -> unit
  val copy : 'a t -> 'a t
  val add : 'a t ->
    key:key -> data:'a -> unit
  val remove : 'a t -> key -> unit
  val find : 'a t -> key -> 'a
  val find_all : 'a t -> key -> 'a list
  val replace : 'a t ->
    key:key -> data:'a -> unit
  val mem : 'a t -> key -> bool
  val iter : f:(key:key -> data:'a -> unit) ->
    'a t -> unit
  val fold : f:(key:key -> data:'a -> 'b -> 'b) ->

```

```

        'a t -> init:'b -> 'b
    val length : 'a t -> int
    val stats : 'a t -> MoreLabels.Hashtbl.statistics
end

module Make :
functor (H : HashedType) -> S with type key = H.t
module MakeSeeded :
functor (H : SeededHashedType) -> SeededS with type key = H.t
val hash : 'a -> int
val seeded_hash : int -> 'a -> int
val hash_param : int -> int -> 'a -> int
val seeded_hash_param : int -> int -> int -> 'a -> int
end

module Map :
sig
    module type OrderedType =
    Map.OrderedType
    module type S =
    sig
        type key
        type +'a t
        val empty : 'a t
        val is_empty : 'a t -> bool
        val mem : key -> 'a t -> bool
        val add : key:key ->
            data:'a -> 'a t -> 'a t
        val singleton : key -> 'a -> 'a t
        val remove : key -> 'a t -> 'a t
        val merge :
            f:(key -> 'a option -> 'b option -> 'c option) ->
            'a t -> 'b t -> 'c t
        val compare : cmp:(('a -> 'a -> int) ->
            'a t -> 'a t -> int)
        val equal : cmp:(('a -> 'a -> bool) ->
            'a t -> 'a t -> bool)
        val iter : f:(key:key -> data:'a -> unit) ->
            'a t -> unit
    end
end

```

```

    val fold : f:(key:key -> data:'a -> 'b -> 'b) ->
      'a t -> init:'b -> 'b
    val for_all : f:(key -> 'a -> bool) -> 'a t -> bool
    val exists : f:(key -> 'a -> bool) -> 'a t -> bool
    val filter : f:(key -> 'a -> bool) ->
      'a t -> 'a t
    val partition : f:(key -> 'a -> bool) ->
      'a t -> 'a t * 'a t
    val cardinal : 'a t -> int
    val bindings : 'a t -> (key * 'a) list
    val min_binding : 'a t -> key * 'a
    val max_binding : 'a t -> key * 'a
    val choose : 'a t -> key * 'a
    val split : key ->
      'a t ->
        'a t * 'a option * 'a t
    val find : key -> 'a t -> 'a
    val map : f:('a -> 'b) -> 'a t -> 'b t
    val mapi : f:(key -> 'a -> 'b) ->
      'a t -> 'b t
  end

  module Make :
    functor (Ord : OrderedType) -> S with type key = Ord.t
  end

  module Set :
    sig
      module type OrderedType =
        Set.OrderedType
      module type S =
        sig
          type elt
          type t
          val empty : t
          val is_empty : t -> bool
          val mem : elt -> t -> bool
          val add : elt -> t -> t
          val singleton : elt -> t
        end
      end
    end
  end

```

```

    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : f:(elt -> unit) -> t -> unit
    val fold : f:(elt -> 'a -> 'a) -> t -> init:'a -> 'a
    val for_all : f:(elt -> bool) -> t -> bool
    val exists : f:(elt -> bool) -> t -> bool
    val filter : f:(elt -> bool) -> t -> t
    val partition : f:(elt -> bool) ->
        t -> t * t
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
    val split : elt ->
        t -> t * bool * t
    val find : elt -> t -> elt
    val of_list : elt list -> t
end

module Make :
    functor (Ord : OrderedType) -> S with type elt = Ord.t
end

```

29 Module Oo : Operations on objects

```
val copy : (< .. > as 'a) -> 'a
```

`Oo.copy o` returns a copy of object `o`, that is a fresh object with the same methods and instance variables as `o`.

```
val id : < .. > -> int
```

Return an integer identifying this object, unique for the current execution of the program. The generic comparison and hashing functions are based on this integer. When an object is obtained by unmarshaling, the id is refreshed, and thus different from the original object. As

a consequence, the internal invariants of data structures such as hash table or sets containing objects are broken after unmarshaling the data structures.

30 Module Gc : Memory management control and statistics; finalised values.

```
type stat = {  
  minor_words : float ;  
    Number of words allocated in the minor heap since the program was started. This  
    number is accurate in byte-code programs, but only an approximation in programs  
    compiled to native code.  
  promoted_words : float ;  
    Number of words allocated in the minor heap that survived a minor collection and  
    were moved to the major heap since the program was started.  
  major_words : float ;  
    Number of words allocated in the major heap, including the promoted words, since  
    the program was started.  
  minor_collections : int ;  
    Number of minor collections since the program was started.  
  major_collections : int ;  
    Number of major collection cycles completed since the program was started.  
  heap_words : int ;  
    Total size of the major heap, in words.  
  heap_chunks : int ;  
    Number of contiguous pieces of memory that make up the major heap.  
  live_words : int ;  
    Number of words of live data in the major heap, including the header words.  
  live_blocks : int ;  
    Number of live blocks in the major heap.  
  free_words : int ;  
    Number of words in the free list.  
  free_blocks : int ;  
    Number of blocks in the free list.  
  largest_free : int ;  
    Size (in words) of the largest block in the free list.  
  fragments : int ;
```

Number of wasted words due to fragmentation. These are 1-words free blocks placed between two live blocks. They are not available for allocation.

`compactions : int ;`

Number of heap compactions since the program was started.

`top_heap_words : int ;`

Maximum size reached by the major heap, in words.

`stack_size : int ;`

Current size of the stack, in words.

Since: 3.12.0

}

The memory management counters are returned in a `stat` record.

The total amount of memory allocated by the program since it was started is (in words)

`minor_words + major_words - promoted_words`. Multiply by the word size (4 on a 32-bit machine, 8 on a 64-bit machine) to get the number of bytes.

`type control = {`

`mutable minor_heap_size : int ;`

The size (in words) of the minor heap. Changing this parameter will trigger a minor collection. Default: 256k.

`mutable major_heap_increment : int ;`

How much to add to the major heap when increasing it. If this number is less than or equal to 1000, it is a percentage of the current heap size (i.e. setting it to 100 will double the heap size at each increase). If it is more than 1000, it is a fixed number of words that will be added to the heap. Default: 15.

`mutable space_overhead : int ;`

The major GC speed is computed from this parameter. This is the memory that will be "wasted" because the GC does not immediately collect unreachable blocks. It is expressed as a percentage of the memory used for live data. The GC will work more (use more CPU time and collect blocks more eagerly) if `space_overhead` is smaller. Default: 80.

`mutable verbose : int ;`

This value controls the GC messages on standard error output. It is a sum of some of the following flags, to print messages on the corresponding events:

- 0x001 Start of major GC cycle.
- 0x002 Minor collection and major GC slice.
- 0x004 Growing and shrinking of the heap.
- 0x008 Resizing of stacks and memory manager tables.
- 0x010 Heap compaction.

- 0x020 Change of GC parameters.
- 0x040 Computation of major GC slice size.
- 0x080 Calling of finalisation functions.
- 0x100 Bytecode executable and shared library search at start-up.
- 0x200 Computation of compaction-triggering condition. Default: 0.

`mutable max_overhead : int ;`

Heap compaction is triggered when the estimated amount of "wasted" memory is more than `max_overhead` percent of the amount of live data. If `max_overhead` is set to 0, heap compaction is triggered at the end of each major GC cycle (this setting is intended for testing purposes only). If `max_overhead` \geq 1000000, compaction is never triggered. If compaction is permanently disabled, it is strongly suggested to set `allocation_policy` to 1. Default: 500.

`mutable stack_limit : int ;`

The maximum size of the stack (in words). This is only relevant to the byte-code runtime, as the native code runtime uses the operating system's stack. Default: 1024k.

`mutable allocation_policy : int ;`

The policy used for allocating in the heap. Possible values are 0 and 1. 0 is the next-fit policy, which is quite fast but can result in fragmentation. 1 is the first-fit policy, which can be slower in some cases but can be better for programs with fragmentation problems. Default: 0.

Since: 3.11.0

}

The GC parameters are given as a `control` record. Note that these parameters can also be initialised by setting the `OCAMLRUNPARAM` environment variable. See the documentation of `ocamlrun`.

`val stat : unit -> stat`

Return the current values of the memory management counters in a `stat` record. This function examines every heap block to get the statistics.

`val quick_stat : unit -> stat`

Same as `stat` except that `live_words`, `live_blocks`, `free_words`, `free_blocks`, `largest_free`, and `fragments` are set to 0. This function is much faster than `stat` because it does not need to go through the heap.

`val counters : unit -> float * float * float`

Return (`minor_words`, `promoted_words`, `major_words`). This function is as fast as `quick_stat`.

`val get : unit -> control`

Return the current values of the GC parameters in a `control` record.

```
val set : control -> unit
```

`set r` changes the GC parameters according to the `control` record `r`. The normal usage is:
`Gc.set { (Gc.get()) with Gc.verbose = 0x00d }`

```
val minor : unit -> unit
```

Trigger a minor collection.

```
val major_slice : int -> int
```

Do a minor collection and a slice of major collection. The argument is the size of the slice, 0 to use the automatically-computed slice size. In all cases, the result is the computed slice size.

```
val major : unit -> unit
```

Do a minor collection and finish the current major collection cycle.

```
val full_major : unit -> unit
```

Do a minor collection, finish the current major collection cycle, and perform a complete new cycle. This will collect all currently unreachable blocks.

```
val compact : unit -> unit
```

Perform a full major collection and compact the heap. Note that heap compaction is a lengthy operation.

```
val print_stat : Pervasives.out_channel -> unit
```

Print the current values of the memory management counters (in human-readable form) into the channel argument.

```
val allocated_bytes : unit -> float
```

Return the total number of bytes allocated since the program was started. It is returned as a `float` to avoid overflow problems with `int` on 32-bit machines.

```
val finalise : ('a -> unit) -> 'a -> unit
```

`finalise f v` registers `f` as a finalisation function for `v`. `v` must be heap-allocated. `f` will be called with `v` as argument at some point between the first time `v` becomes unreachable and the time `v` is collected by the GC. Several functions can be registered for the same value, or even several instances of the same function. Each instance will be called once (or never, if the program terminates before `v` becomes unreachable).

The GC will call the finalisation functions in the order of deallocation. When several values become unreachable at the same time (i.e. during the same GC cycle), the finalisation functions will be called in the reverse order of the corresponding calls to `finalise`. If `finalise` is called in the same order as the values are allocated, that means each value is finalised before the values it depends upon. Of course, this becomes false if additional dependencies are introduced by assignments.

In the presence of multiple OCaml threads it should be assumed that any particular finaliser may be executed in any of the threads.

Anything reachable from the closure of finalisation functions is considered reachable, so the following code will not work as expected:

- `let v = ... in Gc.finalise (fun x -> ... v ...) v`

Instead you should make sure that `v` is not in the closure of the finalisation function by writing:

- `let f = fun x -> ... ;; let v = ... in Gc.finalise f v`

The `f` function can use all features of OCaml, including assignments that make the value reachable again. It can also loop forever (in this case, the other finalisation functions will not be called during the execution of `f`, unless it calls `finalise_release`). It can call `finalise` on `v` or other values to register other functions or even itself. It can raise an exception; in this case the exception will interrupt whatever the program was doing when the function was called.

`finalise` will raise `Invalid_argument` if `v` is not heap-allocated. Some examples of values that are not heap-allocated are integers, constant constructors, booleans, the empty array, the empty list, the unit value. The exact list of what is heap-allocated or not is implementation-dependent. Some constant values can be heap-allocated but never deallocated during the lifetime of the program, for example a list of integer constants; this is also implementation-dependent. You should also be aware that compiler optimisations may duplicate some immutable values, for example floating-point numbers when stored into arrays, so they can be finalised and collected while another copy is still in use by the program.

The results of calling `String.make[44]`, `Bytes.make[12]`, `Bytes.create[12]`, `Array.make[7]`, and `Pervasives.ref[17]` are guaranteed to be heap-allocated and non-constant except when the length argument is 0.

```
val finalise_release : unit -> unit
```

A finalisation function may call `finalise_release` to tell the GC that it can launch the next finalisation function without waiting for the current one to return.

```
type alarm
```

An alarm is a piece of data that calls a user function at the end of each major GC cycle. The following functions are provided to create and delete alarms.

```
val create_alarm : (unit -> unit) -> alarm
```

`create_alarm f` will arrange for `f` to be called at the end of each major GC cycle, starting with the current cycle or the next one. A value of type `alarm` is returned that you can use to call `delete_alarm`.

```
val delete_alarm : alarm -> unit
```

`delete_alarm a` will stop the calls to the function associated to `a`. Calling `delete_alarm a` again has no effect.

31 Module Callback : Registering OCaml values with the C runtime.

This module allows OCaml values to be registered with the C runtime under a symbolic name, so that C code can later call back registered OCaml functions, or raise registered OCaml exceptions.

```
val register : string -> 'a -> unit
```

`Callback.register n v` registers the value `v` under the name `n`. C code can later retrieve a handle to `v` by calling `caml_named_value(n)`.

```
val register_exception : string -> exn -> unit
```

`Callback.register_exception n exn` registers the exception contained in the exception value `exn` under the name `n`. C code can later retrieve a handle to the exception by calling `caml_named_value(n)`. The exception value thus obtained is suitable for passing as first argument to `raise_constant` or `raise_with_arg`.

32 Module Buffer : Extensible buffers.

This module implements buffers that automatically expand as necessary. It provides accumulative concatenation of strings in quasi-linear time (instead of quadratic time when strings are concatenated pairwise).

```
type t
```

The abstract type of buffers.

```
val create : int -> t
```

`create n` returns a fresh buffer, initially empty. The `n` parameter is the initial size of the internal byte sequence that holds the buffer contents. That byte sequence is automatically reallocated when more than `n` characters are stored in the buffer, but shrinks back to `n` characters when `reset` is called. For best performance, `n` should be of the same order of magnitude as the number of characters that are expected to be stored in the buffer (for instance, 80 for a buffer that holds one output line). Nothing bad will happen if the buffer grows beyond that limit, however. In doubt, take `n = 16` for instance. If `n` is not between 1 and `Sys.max_string_length[22]`, it will be clipped to that interval.

```
val contents : t -> string
```

Return a copy of the current contents of the buffer. The buffer itself is unchanged.

```
val to_bytes : t -> bytes
```

Return a copy of the current contents of the buffer. The buffer itself is unchanged.

Since: 4.02

```
val sub : t -> int -> int -> string
```

`Buffer.sub b off len` returns (a copy of) the bytes from the current contents of the buffer `b` starting at offset `off` of length `len` bytes. May raise `Invalid_argument` if out of bounds request. The buffer itself is unaffected.

`val blit : t -> int -> bytes -> int -> int -> unit`

`Buffer.blit src srcoff dst dstoff len` copies `len` characters from the current contents of the buffer `src`, starting at offset `srcoff` to `dst`, starting at character `dstoff`.

Raise `Invalid_argument` if `srcoff` and `len` do not designate a valid range of `src`, or if `dstoff` and `len` do not designate a valid range of `dst`.

Since: 3.11.2

`val nth : t -> int -> char`

get the `n`-th character of the buffer. Raise `Invalid_argument` if index out of bounds

`val length : t -> int`

Return the number of characters currently contained in the buffer.

`val clear : t -> unit`

Empty the buffer.

`val reset : t -> unit`

Empty the buffer and deallocate the internal byte sequence holding the buffer contents, replacing it with the initial internal byte sequence of length `n` that was allocated by `Buffer.create[32] n`. For long-lived buffers that may have grown a lot, `reset` allows faster reclamation of the space used by the buffer.

`val add_char : t -> char -> unit`

`add_char b c` appends the character `c` at the end of the buffer `b`.

`val add_string : t -> string -> unit`

`add_string b s` appends the string `s` at the end of the buffer `b`.

`val add_bytes : t -> bytes -> unit`

`add_string b s` appends the string `s` at the end of the buffer `b`.

Since: 4.02

`val add_substring : t -> string -> int -> int -> unit`

`add_substring b s ofs len` takes `len` characters from offset `ofs` in string `s` and appends them at the end of the buffer `b`.

`val add_subbytes : t -> bytes -> int -> int -> unit`

`add_substring b s ofs len` takes `len` characters from offset `ofs` in byte sequence `s` and appends them at the end of the buffer `b`.

Since: 4.02

```

val add_substitute : t -> (string -> string) -> string -> unit
  add_substitute b f s appends the string pattern s at the end of the buffer b with
  substitution. The substitution process looks for variables into the pattern and substitutes
  each variable name by its value, as obtained by applying the mapping f to the variable
  name. Inside the string pattern, a variable name immediately follows a non-escaped $
  character and is one of the following:
    • a non empty sequence of alphanumeric or _ characters,
    • an arbitrary sequence of characters enclosed by a pair of matching parentheses or curly
      brackets. An escaped $ character is a $ that immediately follows a backslash character;
      it then stands for a plain $. Raise Not_found if the closing character of a parenthesized
      variable cannot be found.

val add_buffer : t -> t -> unit
  add_buffer b1 b2 appends the current contents of buffer b2 at the end of buffer b1. b2 is
  not modified.

val add_channel : t -> Pervasives.in_channel -> int -> unit
  add_channel b ic n reads exactly n character from the input channel ic and stores them
  at the end of buffer b. Raise End_of_file if the channel contains fewer than n characters.

val output_buffer : Pervasives.out_channel -> t -> unit
  output_buffer oc b writes the current contents of buffer b on the output channel oc.

```

33 Module Sort : Sorting and merging lists.

This module is obsolete and exists only for backward compatibility. The sorting functions in `Array`[7] and `List`[16] should be used instead. The new functions are faster and use less memory. Sorting and merging lists.

```

val list : ('a -> 'a -> bool) -> 'a list -> 'a list
  Sort a list in increasing order according to an ordering predicate. The predicate should
  return true if its first argument is less than or equal to its second argument.

val array : ('a -> 'a -> bool) -> 'a array -> unit
  Sort an array in increasing order according to an ordering predicate. The predicate should
  return true if its first argument is less than or equal to its second argument. The array is
  sorted in place.

val merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
  Merge two lists according to the given predicate. Assuming the two argument lists are sorted
  according to the predicate, merge returns a sorted list containing the elements from the two
  lists. The behavior is undefined if the two argument lists were not sorted.

```

34 Module Lazy : Deferred computations.

`type 'a t = 'a lazy_t`

A value of type `'a Lazy.t` is a deferred computation, called a suspension, that has a result of type `'a`. The special expression syntax `lazy (expr)` makes a suspension of the computation of `expr`, without computing `expr` itself yet. "Forcing" the suspension will then compute `expr` and return its result.

Note: `lazy_t` is the built-in type constructor used by the compiler for the `lazy` keyword. You should not use it directly. Always use `Lazy.t` instead.

Note: `Lazy.force` is not thread-safe. If you use this module in a multi-threaded program, you will need to add some locks.

Note: if the program is compiled with the `-rectypes` option, ill-founded recursive definitions of the form `let rec x = lazy x` or `let rec x = lazy(lazy(...(lazy x)))` are accepted by the type-checker and lead, when forced, to ill-formed values that trigger infinite loops in the garbage collector and other parts of the run-time system. Without the `-rectypes` option, such ill-founded recursive definitions are rejected by the type-checker.

`exception Undefined`

`val force : 'a t -> 'a`

`force x` forces the suspension `x` and returns its result. If `x` has already been forced, `Lazy.force x` returns the same value again without recomputing it. If it raised an exception, the same exception is raised again. Raise `Undefined` if the forcing of `x` tries to force `x` itself recursively.

`val force_val : 'a t -> 'a`

`force_val x` forces the suspension `x` and returns its result. If `x` has already been forced, `force_val x` returns the same value again without recomputing it. Raise `Undefined` if the forcing of `x` tries to force `x` itself recursively. If the computation of `x` raises an exception, it is unspecified whether `force_val x` raises the same exception or `Undefined`.

`val from_fun : (unit -> 'a) -> 'a t`

`from_fun f` is the same as `lazy (f ())` but slightly more efficient.

Since: 4.00.0

`val from_val : 'a -> 'a t`

`from_val v` returns an already-forced suspension of `v`. This is for special purposes only and should not be confused with `lazy (v)`.

Since: 4.00.0

`val is_val : 'a t -> bool`

`is_val x` returns `true` if `x` has already been forced and did not raise an exception.

Since: 4.00.0

```

val lazy_from_fun : (unit -> 'a) -> 'a t
    Deprecated. synonym for from_fun.

val lazy_from_val : 'a -> 'a t
    Deprecated. synonym for from_val.

val lazy_is_val : 'a t -> bool
    Deprecated. synonym for is_val.

```

35 Module Camlinternal00 : Run-time support for objects and classes.

All functions in this module are for system use only, not for the casual user.

Classes

```

type tag
type label
type table
type meth
type t
type obj
type closure
val public_method_label : string -> tag
val new_method : table -> label
val new_variable : table -> string -> int
val new_methods_variables :
    table ->
    string array -> string array -> label array
val get_variable : table -> string -> int
val get_variables : table -> string array -> int array
val get_method_label : table -> string -> label
val get_method_labels : table -> string array -> label array
val get_method : table -> label -> meth
val set_method : table -> label -> meth -> unit
val set_methods : table -> label array -> unit
val narrow : table -> string array -> string array -> string array -> unit
val widen : table -> unit
val add_initializer : table -> (obj -> unit) -> unit
val dummy_table : table
val create_table : string array -> table

```

```

val init_class : table -> unit
val inherits :
  table ->
  string array ->
  string array ->
  string array ->
  t * (table -> obj -> Obj.t) *
  t * obj -> bool -> Obj.t array
val make_class :
  string array ->
  (table -> Obj.t -> t) ->
  t * (table -> Obj.t -> t) *
  (Obj.t -> t) * Obj.t
type init_table
val make_class_store : string array ->
  (table -> t) ->
  init_table -> unit
val dummy_class :
  string * int * int ->
  t * (table -> Obj.t -> t) *
  (Obj.t -> t) * Obj.t
  Objects
val copy : (< .. > as 'a) -> 'a
val create_object : table -> obj
val create_object_opt : obj -> table -> obj
val run_initializers : obj -> table -> unit
val run_initializers_opt : obj ->
  obj -> table -> obj
val create_object_and_run_initializers : obj -> table -> obj
val send : obj -> tag -> t
val sendcache : obj ->
  tag -> t -> int -> t
val sendself : obj -> label -> t
val get_public_method : obj -> tag -> closure
  Table cache
type tables
val lookup_tables : tables ->
  closure array -> tables
  Builtins to reduce code size
type impl =
  | GetConst
  | GetVar

```

```

| GetEnv
| GetMeth
| SetVar
| AppConst
| AppVar
| AppEnv
| AppMeth
| AppConstConst
| AppConstVar
| AppConstEnv
| AppConstMeth
| AppVarConst
| AppEnvConst
| AppMethConst
| MethAppConst
| MethAppVar
| MethAppEnv
| MethAppMeth
| SendConst
| SendVar
| SendEnv
| SendMeth
| Closure of closure
Parameters
type params = {
  mutable compact_table : bool ;
  mutable copy_parent : bool ;
  mutable clean_when_copying : bool ;
  mutable retry_count : int ;
  mutable bucket_small_size : int ;
}
val params : params
Statistics
type stats = {
  classes : int ;
  methods : int ;
  inst_vars : int ;
}
val stats : unit -> stats

```

36 Module Lexing : The run-time library for lexers generated by ocamllex.

Positions

```
type position = {  
  pos_fname : string ;  
  pos_lnum : int ;  
  pos_bol : int ;  
  pos_cnum : int ;  
}
```

A value of type `position` describes a point in a source file. `pos_fname` is the file name; `pos_lnum` is the line number; `pos_bol` is the offset of the beginning of the line (number of characters between the beginning of the lexbuf and the beginning of the line); `pos_cnum` is the offset of the position (number of characters between the beginning of the lexbuf and the position). The difference between `pos_cnum` and `pos_bol` is the character offset within the line (i.e. the column number, assuming each character is one column wide).

See the documentation of type `lexbuf` for information about how the lexing engine will manage positions.

```
val dummy_pos : position
```

A value of type `position`, guaranteed to be different from any valid position.

Lexer buffers

```
type lexbuf = {  
  refill_buff : lexbuf -> unit ;  
  mutable lex_buffer : bytes ;  
  mutable lex_buffer_len : int ;  
  mutable lex_abs_pos : int ;  
  mutable lex_start_pos : int ;  
  mutable lex_curr_pos : int ;  
  mutable lex_last_pos : int ;  
  mutable lex_last_action : int ;  
  mutable lex_eof_reached : bool ;  
  mutable lex_mem : int array ;  
  mutable lex_start_p : position ;  
  mutable lex_curr_p : position ;  
}
```

The type of lexer buffers. A lexer buffer is the argument passed to the scanning functions defined by the generated scanners. The lexer buffer holds the current state of the scanner, plus a function to refill the buffer from the input.

At each token, the lexing engine will copy `lex_curr_p` to `lex_start_p`, then change the `pos_cnum` field of `lex_curr_p` by updating it with the number of characters read since the start of the `lexbuf`. The other fields are left unchanged by the lexing engine. In order to

keep them accurate, they must be initialised before the first use of the `lexbuf`, and updated by the relevant lexer actions (i.e. at each end of line – see also `new_line`).

`val from_channel : Pervasives.in_channel -> lexbuf`

Create a lexer buffer on the given input channel. `Lexing.from_channel inchan` returns a lexer buffer which reads from the input channel `inchan`, at the current reading position.

`val from_string : string -> lexbuf`

Create a lexer buffer which reads from the given string. Reading starts from the first character in the string. An end-of-input condition is generated when the end of the string is reached.

`val from_function : (bytes -> int -> int) -> lexbuf`

Create a lexer buffer with the given function as its reading method. When the scanner needs more characters, it will call the given function, giving it a byte sequence `s` and a byte count `n`. The function should put `n` bytes or fewer in `s`, starting at index 0, and return the number of bytes provided. A return value of 0 means end of input.

Functions for lexer semantic actions

Functions for lexer semantic actions

The following functions can be called from the semantic actions of lexer definitions (the ML code enclosed in braces that computes the value returned by lexing functions). They give access to the character string matched by the regular expression associated with the semantic action. These functions must be applied to the argument `lexbuf`, which, in the code generated by `ocamllex`, is bound to the lexer buffer passed to the parsing function.

`val lexeme : lexbuf -> string`

`Lexing.lexeme lexbuf` returns the string matched by the regular expression.

`val lexeme_char : lexbuf -> int -> char`

`Lexing.lexeme_char lexbuf i` returns character number `i` in the matched string.

`val lexeme_start : lexbuf -> int`

`Lexing.lexeme_start lexbuf` returns the offset in the input stream of the first character of the matched string. The first character of the stream has offset 0.

`val lexeme_end : lexbuf -> int`

`Lexing.lexeme_end lexbuf` returns the offset in the input stream of the character following the last character of the matched string. The first character of the stream has offset 0.

`val lexeme_start_p : lexbuf -> position`

Like `lexeme_start`, but return a complete `position` instead of an offset.

`val lexeme_end_p : lexbuf -> position`

Like `lexeme_end`, but return a complete `position` instead of an offset.

`val new_line : lexbuf -> unit`

Update the `lex_curr_p` field of the `lexbuf` to reflect the start of a new line. You can call this function in the semantic action of the rule that matches the end-of-line character.

Since: 3.11.0

Miscellaneous functions

`val flush_input : lexbuf -> unit`

Discard the contents of the buffer and reset the current position to 0. The next use of the `lexbuf` will trigger a refill.

37 Module Nativeint : Processor-native integers.

This module provides operations on the type `nativeint` of signed 32-bit integers (on 32-bit platforms) or signed 64-bit integers (on 64-bit platforms). This integer type has exactly the same width as that of a pointer type in the C compiler. All arithmetic operations over `nativeint` are taken modulo 2^{32} or 2^{64} depending on the word size of the architecture.

Performance notice: values of type `nativeint` occupy more memory space than values of type `int`, and arithmetic operations on `nativeint` are generally slower than those on `int`. Use `nativeint` only when the application requires the extra bit of precision over the `int` type.

`val zero : nativeint`

The native integer 0.

`val one : nativeint`

The native integer 1.

`val minus_one : nativeint`

The native integer -1.

`val neg : nativeint -> nativeint`

Unary negation.

`val add : nativeint -> nativeint -> nativeint`

Addition.

`val sub : nativeint -> nativeint -> nativeint`

Subtraction.

`val mul : nativeint -> nativeint -> nativeint`

Multiplication.

`val div : nativeint -> nativeint -> nativeint`

Integer division. Raise `Division_by_zero` if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for `Pervasives.(/)`[17].

`val rem : nativeint -> nativeint -> nativeint`

Integer remainder. If `y` is not zero, the result of `Nativeint.rem x y` satisfies the following properties: `Nativeint.zero <= Nativeint.rem x y < Nativeint.abs y` and `x = Nativeint.add (Nativeint.mul (Nativeint.div x y) y) (Nativeint.rem x y)`. If `y = 0`, `Nativeint.rem x y` raises `Division_by_zero`.

`val succ : nativeint -> nativeint`

Successor. `Nativeint.succ x` is `Nativeint.add x Nativeint.one`.

`val pred : nativeint -> nativeint`

Predecessor. `Nativeint.pred x` is `Nativeint.sub x Nativeint.one`.

`val abs : nativeint -> nativeint`

Return the absolute value of its argument.

`val size : int`

The size in bits of a native integer. This is equal to 32 on a 32-bit platform and to 64 on a 64-bit platform.

`val max_int : nativeint`

The greatest representable native integer, either $2^{31} - 1$ on a 32-bit platform, or $2^{63} - 1$ on a 64-bit platform.

`val min_int : nativeint`

The greatest representable native integer, either -2^{31} on a 32-bit platform, or -2^{63} on a 64-bit platform.

`val logand : nativeint -> nativeint -> nativeint`

Bitwise logical and.

`val logor : nativeint -> nativeint -> nativeint`

Bitwise logical or.

`val logxor : nativeint -> nativeint -> nativeint`

Bitwise logical exclusive or.

`val lognot : nativeint -> nativeint`

Bitwise logical negation

`val shift_left : nativeint -> int -> nativeint`

`Nativeint.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= bitsize`, where `bitsize` is 32 on a 32-bit platform and 64 on a 64-bit platform.

```

val shift_right : nativeint -> int -> nativeint
    Nativeint.shift_right x y shifts x to the right by y bits. This is an arithmetic shift: the
    sign bit of x is replicated and inserted in the vacated bits. The result is unspecified if y < 0
    or y >= bitsize.

val shift_right_logical : nativeint -> int -> nativeint
    Nativeint.shift_right_logical x y shifts x to the right by y bits. This is a logical shift:
    zeroes are inserted in the vacated bits regardless of the sign of x. The result is unspecified if
    y < 0 or y >= bitsize.

val of_int : int -> nativeint
    Convert the given integer (type int) to a native integer (type nativeint).

val to_int : nativeint -> int
    Convert the given native integer (type nativeint) to an integer (type int). The high-order
    bit is lost during the conversion.

val of_float : float -> nativeint
    Convert the given floating-point number to a native integer, discarding the fractional part
    (truncate towards 0). The result of the conversion is undefined if, after truncation, the
    number is outside the range [Nativeint.min_int[37], Nativeint.max_int[37]].

val to_float : nativeint -> float
    Convert the given native integer to a floating-point number.

val of_int32 : int32 -> nativeint
    Convert the given 32-bit integer (type int32) to a native integer.

val to_int32 : nativeint -> int32
    Convert the given native integer to a 32-bit integer (type int32). On 64-bit platforms, the
    64-bit native integer is taken modulo  $2^{32}$ , i.e. the top 32 bits are lost. On 32-bit platforms,
    the conversion is exact.

val of_string : string -> nativeint
    Convert the given string to a native integer. The string is read in decimal (by default) or in
    hexadecimal, octal or binary if the string begins with 0x, 0o or 0b respectively. Raise
    Failure "int_of_string" if the given string is not a valid representation of an integer, or if
    the integer represented exceeds the range of integers representable in type nativeint.

val to_string : nativeint -> string
    Return the string representation of its argument, in decimal.

type t = nativeint
    An alias for the type of native integers.

```

```
val compare : t -> t -> int
```

The comparison function for native integers, with the same specification as `Pervasives.compare`[17]. Along with the type `t`, this function `compare` allows the module `Nativeint` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

Deprecated functions

```
val format : string -> nativeint -> string
```

`Nativeint.format fmt n` return the string representation of the native integer `n` in the format specified by `fmt`. `fmt` is a `Printf`-style format consisting of exactly one `%d`, `%i`, `%u`, `%x`, `%X` or `%o` conversion specification. This function is deprecated; use `Printf.sprintf`[24] with a `%nx` format instead.

38 Module Set : Sets over ordered types.

This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.

The `Make` functor constructs implementations for any type, given a `compare` function. For instance:

```
module IntPairs =
  struct
    type t = int * int
    let compare (x0,y0) (x1,y1) =
      match Pervasives.compare x0 x1 with
      | 0 -> Pervasives.compare y0 y1
      | c -> c
  end

module PairsSet = Set.Make(IntPairs)

let m = PairsSet.(empty |> add (2,3) |> add (5,7) |> add (11,13))
```

This creates a new module `PairsSet`, with a new type `PairsSet.t` of sets of `int * int`.

```
module type OrderedType =
```

```
sig
```

```
  type t
```

The type of the set elements.

```
  val compare : t -> t -> int
```

A total ordering function over the set elements. This is a two-argument function **f** such that **f e1 e2** is zero if the elements **e1** and **e2** are equal, **f e1 e2** is strictly negative if **e1** is smaller than **e2**, and **f e1 e2** is strictly positive if **e1** is greater than **e2**. Example: a suitable ordering function is the generic structural comparison function `Pervasives.compare`[17].

end

Input signature of the functor `Set.Make`[38].

```
module type S =
  sig
    type elt
      The type of the set elements.

    type t
      The type of sets.

    val empty : t
      The empty set.

    val is_empty : t -> bool
      Test whether a set is empty or not.

    val mem : elt -> t -> bool
      mem x s tests whether x belongs to the set s.

    val add : elt -> t -> t
      add x s returns a set containing all elements of s, plus x. If x was already in s, s is
      returned unchanged.

    val singleton : elt -> t
      singleton x returns the one-element set containing only x.

    val remove : elt -> t -> t
      remove x s returns a set containing all elements of s, except x. If x was not in s, s is
      returned unchanged.

    val union : t -> t -> t
      Set union.

    val inter : t -> t -> t
```

Set intersection.

```
val diff : t -> t -> t
```

Set difference.

```
val compare : t -> t -> int
```

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

```
val equal : t -> t -> bool
```

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

```
val subset : t -> t -> bool
```

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

```
val iter : (elt -> unit) -> t -> unit
```

`iter f s` applies `f` in turn to all elements of `s`. The elements of `s` are presented to `f` in increasing order with respect to the ordering over the type of the elements.

```
val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`, in increasing order.

```
val for_all : (elt -> bool) -> t -> bool
```

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

```
val exists : (elt -> bool) -> t -> bool
```

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

```
val filter : (elt -> bool) -> t -> t
```

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

```
val partition : (elt -> bool) -> t -> t * t
```

`partition p s` returns a pair of sets (`s1`, `s2`), where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

```
val cardinal : t -> int
```

Return the number of elements of a set.

```
val elements : t -> elt list
```

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Ord.compare`, where `Ord` is the argument given to `Set.Make[38]`.

```
val min_elt : t -> elt
```

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

```
val max_elt : t -> elt
```

Same as `Set.S.min_elt[38]`, but returns the largest element of the given set.

```
val choose : t -> elt
```

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
val split : elt -> t -> t * bool * t
```

`split x s` returns a triple `(l, present, r)`, where `l` is the set of elements of `s` that are strictly less than `x`; `r` is the set of elements of `s` that are strictly greater than `x`; `present` is `false` if `s` contains no element equal to `x`, or `true` if `s` contains an element equal to `x`.

```
val find : elt -> t -> elt
```

`find x s` returns the element of `s` equal to `x` (according to `Ord.compare`), or raise `Not_found` if no such element exists.

Since: 4.01.0

```
val of_list : elt list -> t
```

`of_list l` creates a set from a list of elements. This is usually more efficient than folding `add` over the list, except perhaps for lists with many duplicated elements.

Since: 4.02.0

end

Output signature of the functor `Set.Make[38]`.

```
module Make :
```

```
  functor (Ord : OrderedType) -> S with type elt = Ord.t
```

Functor building an implementation of the set structure given a totally ordered type.

39 Module Format : Pretty printing.

This module implements a pretty-printing facility to format text within 'pretty-printing boxes'. The pretty-printer breaks lines at specified break hints, and indents lines according to the box structure.

For a gentle introduction to the basics of pretty-printing using `Format`, read

<http://caml.inria.fr/resources/doc/guides/format.en.html>[\[http://caml.inria.fr/resources/doc/guides/format.en.html\]](http://caml.inria.fr/resources/doc/guides/format.en.html).

You may consider this module as providing an extension to the `printf` facility to provide automatic line breaking. The addition of pretty-printing annotations to your regular `printf` formats gives you fancy indentation and line breaks. Pretty-printing annotations are described below in the documentation of the function `Format.fprintf`[\[39\]](#).

You may also use the explicit box management and printing functions provided by this module. This style is more basic but more verbose than the `fprintf` concise formats.

For instance, the sequence `open_box 0; print_string "x ="; print_space (); print_int 1; close_box (); print_newline ()` that prints `x = 1` within a pretty-printing box, can be abbreviated as `printf "@[%s@ %i@]@." "x =" 1`, or even shorter `printf "@[x =@ %i@]@." 1`.

Rule of thumb for casual users of this library:

- use simple boxes (as obtained by `open_box 0`);
- use simple break hints (as obtained by `print_cut ()` that outputs a simple break hint, or by `print_space ()` that outputs a space indicating a break hint);
- once a box is opened, display its material with basic printing functions (e. g. `print_int` and `print_string`);
- when the material for a box has been printed, call `close_box ()` to close the box;
- at the end of your routine, flush the pretty-printer to display all the remaining material, e.g. evaluate `print_newline ()`.

The behaviour of pretty-printing commands is unspecified if there is no opened pretty-printing box. Each box opened via one of the `open_` functions below must be closed using `close_box` for proper formatting. Otherwise, some of the material printed in the boxes may not be output, or may be formatted incorrectly.

In case of interactive use, the system closes all opened boxes and flushes all pending text (as with the `print_newline` function) after each phrase. Each phrase is therefore executed in the initial state of the pretty-printer.

Warning: the material output by the following functions is delayed in the pretty-printer queue in order to compute the proper line breaking. Hence, you should not mix calls to the printing functions of the basic I/O system with calls to the functions of this module: this could result in some strange output seemingly unrelated with the evaluation order of printing commands.

Boxes

```
val open_box : int -> unit
```

`open_box d` opens a new pretty-printing box with offset `d`. This box is the general purpose pretty-printing box. Material in this box is displayed 'horizontal or vertical': break hints

inside the box may lead to a new line, if there is no more room on the line to print the remainder of the box, or if a new line may lead to a new indentation (demonstrating the indentation of the box). When a new line is printed in the box, `d` is added to the current indentation.

```
val close_box : unit -> unit
```

Closes the most recently opened pretty-printing box.

Formatting functions

```
val print_string : string -> unit
```

`print_string str` prints `str` in the current box.

```
val print_as : int -> string -> unit
```

`print_as len str` prints `str` in the current box. The pretty-printer formats `str` as if it were of length `len`.

```
val print_int : int -> unit
```

Prints an integer in the current box.

```
val print_float : float -> unit
```

Prints a floating point number in the current box.

```
val print_char : char -> unit
```

Prints a character in the current box.

```
val print_bool : bool -> unit
```

Prints a boolean in the current box.

Break hints

```
val print_space : unit -> unit
```

`print_space ()` is used to separate items (typically to print a space between two words). It indicates that the line may be split at this point. It either prints one space or splits the line. It is equivalent to `print_break 1 0`.

```
val print_cut : unit -> unit
```

`print_cut ()` is used to mark a good break position. It indicates that the line may be split at this point. It either prints nothing or splits the line. This allows line splitting at the current point, without printing spaces or adding indentation. It is equivalent to `print_break 0 0`.

```
val print_break : int -> int -> unit
```

Inserts a break hint in a pretty-printing box. `print_break nspaces offset` indicates that the line may be split (a newline character is printed) at this point, if the contents of the current box does not fit on the current line. If the line is split at that point, `offset` is added to the current indentation. If the line is not split, `nspaces` spaces are printed.

val print_flush : unit -> unit

Flushes the pretty printer: all opened boxes are closed, and all pending text is displayed.

val print_newline : unit -> unit

Equivalent to **print_flush** followed by a new line.

val force_newline : unit -> unit

Forces a newline in the current box. Not the normal way of pretty-printing, you should prefer break hints.

val print_if_newline : unit -> unit

Executes the next formatting command if the preceding line has just been split. Otherwise, ignore the next formatting command.

Margin

val set_margin : int -> unit

set_margin d sets the value of the right margin to **d** (in characters): this value is used to detect line overflows that leads to split lines. Nothing happens if **d** is smaller than 2. If **d** is too large, the right margin is set to the maximum admissible value (which is greater than 10^9).

val get_margin : unit -> int

Returns the position of the right margin.

Maximum indentation limit

val set_max_indent : int -> unit

set_max_indent d sets the value of the maximum indentation limit to **d** (in characters): once this limit is reached, boxes are rejected to the left, if they do not fit on the current line. Nothing happens if **d** is smaller than 2. If **d** is too large, the limit is set to the maximum admissible value (which is greater than 10^9).

val get_max_indent : unit -> int

Return the value of the maximum indentation limit (in characters).

Formatting depth: maximum number of boxes allowed before ellipsis

val set_max_boxes : int -> unit

set_max_boxes max sets the maximum number of boxes simultaneously opened. Material inside boxes nested deeper is printed as an ellipsis (more precisely as the text returned by **get_ellipsis_text ()**). Nothing happens if **max** is smaller than 2.

val get_max_boxes : unit -> int

Returns the maximum number of boxes allowed before ellipsis.

val over_max_boxes : unit -> bool

Tests if the maximum number of boxes allowed have already been opened.

Advanced formatting

`val open_hbox : unit -> unit`

`open_hbox ()` opens a new pretty-printing box. This box is 'horizontal': the line is not split in this box (new lines may still occur inside boxes nested deeper).

`val open_vbox : int -> unit`

`open_vbox d` opens a new pretty-printing box with offset `d`. This box is 'vertical': every break hint inside this box leads to a new line. When a new line is printed in the box, `d` is added to the current indentation.

`val open_hvbox : int -> unit`

`open_hvbox d` opens a new pretty-printing box with offset `d`. This box is 'horizontal-vertical': it behaves as an 'horizontal' box if it fits on a single line, otherwise it behaves as a 'vertical' box. When a new line is printed in the box, `d` is added to the current indentation.

`val open_hovbox : int -> unit`

`open_hovbox d` opens a new pretty-printing box with offset `d`. This box is 'horizontal or vertical': break hints inside this box may lead to a new line, if there is no more room on the line to print the remainder of the box. When a new line is printed in the box, `d` is added to the current indentation.

Tabulations

`val open_tbox : unit -> unit`

Opens a tabulation box.

`val close_tbox : unit -> unit`

Closes the most recently opened tabulation box.

`val print_tbreak : int -> int -> unit`

Break hint in a tabulation box. `print_tbreak spaces offset` moves the insertion point to the next tabulation (`spaces` being added to this position). Nothing occurs if insertion point is already on a tabulation mark. If there is no next tabulation on the line, then a newline is printed and the insertion point moves to the first tabulation of the box. If a new line is printed, `offset` is added to the current indentation.

`val set_tab : unit -> unit`

Sets a tabulation mark at the current insertion point.

`val print_tab : unit -> unit`

`print_tab ()` is equivalent to `print_tbreak 0 0`.

Ellipsis

`val set_ellipsis_text : string -> unit`

Set the text of the ellipsis printed when too many boxes are opened (a single dot, ., by default).

```
val get_ellipsis_text : unit -> string
```

Return the text of the ellipsis.

Semantics Tags

```
type tag = string
```

Semantics tags (or simply *tags*) are used to decorate printed entities for user's defined purposes, e.g. setting font and giving size indications for a display device, or marking delimitation of semantics entities (e.g. HTML or TeX elements or terminal escape sequences).

By default, those tags do not influence line breaking calculation: the tag 'markers' are not considered as part of the printing material that drives line breaking (in other words, the length of those strings is considered as zero for line breaking).

Thus, tag handling is in some sense transparent to pretty-printing and does not interfere with usual indentation. Hence, a single pretty printing routine can output both simple 'verbatim' material or richer decorated output depending on the treatment of tags. By default, tags are not active, hence the output is not decorated with tag information. Once `set_tags` is set to `true`, the pretty printer engine honours tags and decorates the output accordingly.

When a tag has been opened (or closed), it is both and successively 'printed' and 'marked'. Printing a tag means calling a formatter specific function with the name of the tag as argument: that 'tag printing' function can then print any regular material to the formatter (so that this material is enqueued as usual in the formatter queue for further line-breaking computation). Marking a tag means to output an arbitrary string (the 'tag marker'), directly into the output device of the formatter. Hence, the formatter specific 'tag marking' function must return the tag marker string associated to its tag argument. Being flushed directly into the output device of the formatter, tag marker strings are not considered as part of the printing material that drives line breaking (in other words, the length of the strings corresponding to tag markers is considered as zero for line breaking). In addition, advanced users may take advantage of the specificity of tag markers to be precisely output when the pretty printer has already decided where to break the lines, and precisely when the queue is flushed into the output device.

In the spirit of HTML tags, the default tag marking functions output tags enclosed in "<" and ">": hence, the opening marker of tag `t` is "<t>" and the closing marker "</t>".

Default tag printing functions just do nothing.

Tag marking and tag printing functions are user definable and can be set by calling `set_formatter_tag_func`

```
val open_tag : tag -> unit
```

`open_tag t` opens the tag named `t`; the `print_open_tag` function of the formatter is called with `t` as argument; the tag marker `mark_open_tag t` will be flushed into the output device of the formatter.

```
val close_tag : unit -> unit
```

`close_tag ()` closes the most recently opened tag `t`. In addition, the `print_close_tag` function of the formatter is called with `t` as argument. The marker `mark_close_tag t` will be flushed into the output device of the formatter.

```
val set_tags : bool -> unit
```

`set_tags b` turns on or off the treatment of tags (default is off).

```
val set_print_tags : bool -> unit
```

`set_print_tags b` turns on or off the printing of tags.

```
val set_mark_tags : bool -> unit
```

`set_mark_tags b` turns on or off the output of tag markers.

```
val get_print_tags : unit -> bool
```

Return the current status of tags printing.

```
val get_mark_tags : unit -> bool
```

Return the current status of tags marking.

Redirecting the standard formatter output

```
val set_formatter_out_channel : Pervasives.out_channel -> unit
```

Redirect the pretty-printer output to the given channel. (All the output functions of the standard formatter are set to the default output functions printing to the given channel.)

```
val set_formatter_output_functions :
```

`(string -> int -> int -> unit) -> (unit -> unit) -> unit`

`set_formatter_output_functions out flush` redirects the pretty-printer output functions to the functions `out` and `flush`.

The `out` function performs all the pretty-printer string output. It is called with a string `s`, a start position `p`, and a number of characters `n`; it is supposed to output characters `p` to `p + n - 1` of `s`.

The `flush` function is called whenever the pretty-printer is flushed (via conversion `%!` , or pretty-printing indications `@?` or `@.` , or using low level functions `print_flush` or `print_newline`).

```
val get_formatter_output_functions :
```

`unit -> (string -> int -> int -> unit) * (unit -> unit)`

Return the current output functions of the pretty-printer.

Changing the meaning of standard formatter pretty printing

Changing the meaning of standard formatter pretty printing

The `Format` module is versatile enough to let you completely redefine the meaning of pretty printing: you may provide your own functions to define how to handle indentation, line breaking, and even printing of all the characters that have to be printed!

```
type formatter_out_functions = {
  out_string : string -> int -> int -> unit ;
  out_flush  : unit -> unit ;
  out_newline : unit -> unit ;
  out_spaces : int -> unit ;
}
```

```
val set_formatter_out_functions : formatter_out_functions -> unit
```

`set_formatter_out_functions f` Redirect the pretty-printer output to the functions `f.out_string` and `f.out_flush` as described in `set_formatter_output_functions`. In addition, the pretty-printer function that outputs a newline is set to the function `f.out_newline` and the function that outputs indentation spaces is set to the function `f.out_spaces`.

This way, you can change the meaning of indentation (which can be something else than just printing space characters) and the meaning of new lines opening (which can be connected to any other action needed by the application at hand). The two functions `f.out_spaces` and `f.out_newline` are normally connected to `f.out_string` and `f.out_flush`: respective default values for `f.out_space` and `f.out_newline` are `f.out_string (String.make n ' ') 0 n` and `f.out_string "\n" 0 1`.

```
val get_formatter_out_functions : unit -> formatter_out_functions
```

Return the current output functions of the pretty-printer, including line breaking and indentation functions. Useful to record the current setting and restore it afterwards.

Changing the meaning of printing semantics tags

```
type formatter_tag_functions = {
  mark_open_tag : tag -> string ;
  mark_close_tag : tag -> string ;
  print_open_tag : tag -> unit ;
  print_close_tag : tag -> unit ;
}
```

The tag handling functions specific to a formatter: **mark** versions are the 'tag marking' functions that associate a string marker to a tag in order for the pretty-printing engine to flush those markers as 0 length tokens in the output device of the formatter. **print** versions are the 'tag printing' functions that can perform regular printing when a tag is closed or opened.

```
val set_formatter_tag_functions : formatter_tag_functions -> unit
```

`set_formatter_tag_functions tag_funs` changes the meaning of opening and closing tags to use the functions in `tag_funs`.

When opening a tag name `t`, the string `t` is passed to the opening tag marking function (the `mark_open_tag` field of the record `tag_funs`), that must return the opening tag marker for that name. When the next call to `close_tag ()` happens, the tag name `t` is sent back to the closing tag marking function (the `mark_close_tag` field of record `tag_funs`), that must return a closing tag marker for that name.

The `print_` field of the record contains the functions that are called at tag opening and tag closing time, to output regular material in the pretty-printer queue.

```
val get_formatter_tag_functions : unit -> formatter_tag_functions
```

Return the current tag functions of the pretty-printer.

Multiple formatted output

```
type formatter
```

Abstract data corresponding to a pretty-printer (also called a formatter) and all its machinery.

Defining new pretty-printers permits unrelated output of material in parallel on several output channels. All the parameters of a pretty-printer are local to this pretty-printer: margin, maximum indentation limit, maximum number of boxes simultaneously opened, ellipsis, and so on, are specific to each pretty-printer and may be fixed independently. Given a `Pervasives.out_channel` output channel `oc`, a new formatter writing to that channel is simply obtained by calling `formatter_of_out_channel oc`. Alternatively, the `make_formatter` function allocates a new formatter with explicit output and flushing functions (convenient to output material to strings for instance).

```
val formatter_of_out_channel : Pervasives.out_channel -> formatter
```

`formatter_of_out_channel oc` returns a new formatter that writes to the corresponding channel `oc`.

```
val std_formatter : formatter
```

The standard formatter used by the formatting functions above. It is defined as `formatter_of_out_channel stdout`.

```
val err_formatter : formatter
```

A formatter to use with formatting functions below for output to standard error. It is defined as `formatter_of_out_channel stderr`.

```
val formatter_of_buffer : Buffer.t -> formatter
```

`formatter_of_buffer b` returns a new formatter writing to buffer `b`. As usual, the formatter has to be flushed at the end of pretty printing, using `pp_print_flush` or `pp_print_newline`, to display all the pending material.

```
val stdbuf : Buffer.t
```

The string buffer in which `str_formatter` writes.

```
val str_formatter : formatter
```

A formatter to use with formatting functions below for output to the `stdbuf` string buffer. `str_formatter` is defined as `formatter_of_buffer stdbuf`.

```
val flush_str_formatter : unit -> string
```

Returns the material printed with `str_formatter`, flushes the formatter and resets the corresponding buffer.

```
val make_formatter :
```

```
(string -> int -> int -> unit) -> (unit -> unit) -> formatter
```

`make_formatter out flush` returns a new formatter that writes according to the output function `out`, and the flushing function `flush`. For instance, a formatter to the `Pervasives.out_channel oc` is returned by `make_formatter (Pervasives.output oc) (fun () -> Pervasives.flush oc)`.

Basic functions to use with formatters

```
val pp_open_hbox : formatter -> unit -> unit
val pp_open_vbox : formatter -> int -> unit
val pp_open_hvbox : formatter -> int -> unit
val pp_open_hovbox : formatter -> int -> unit
val pp_open_box : formatter -> int -> unit
val pp_close_box : formatter -> unit -> unit
val pp_open_tag : formatter -> string -> unit
val pp_close_tag : formatter -> unit -> unit
val pp_print_string : formatter -> string -> unit
val pp_print_as : formatter -> int -> string -> unit
val pp_print_int : formatter -> int -> unit
val pp_print_float : formatter -> float -> unit
val pp_print_char : formatter -> char -> unit
val pp_print_bool : formatter -> bool -> unit
val pp_print_break : formatter -> int -> int -> unit
val pp_print_cut : formatter -> unit -> unit
val pp_print_space : formatter -> unit -> unit
val pp_force_newline : formatter -> unit -> unit
val pp_print_flush : formatter -> unit -> unit
val pp_print_newline : formatter -> unit -> unit
val pp_print_if_newline : formatter -> unit -> unit
val pp_open_tbox : formatter -> unit -> unit
val pp_close_tbox : formatter -> unit -> unit
val pp_print_tbreak : formatter -> int -> int -> unit
val pp_set_tab : formatter -> unit -> unit
val pp_print_tab : formatter -> unit -> unit
val pp_set_tags : formatter -> bool -> unit
val pp_set_print_tags : formatter -> bool -> unit
val pp_set_mark_tags : formatter -> bool -> unit
val pp_get_print_tags : formatter -> unit -> bool
val pp_get_mark_tags : formatter -> unit -> bool
val pp_set_margin : formatter -> int -> unit
val pp_get_margin : formatter -> unit -> int
val pp_set_max_indent : formatter -> int -> unit
val pp_get_max_indent : formatter -> unit -> int
val pp_set_max_boxes : formatter -> int -> unit
val pp_get_max_boxes : formatter -> unit -> int
```

```

val pp_over_max_boxes : formatter -> unit -> bool
val pp_set_ellipsis_text : formatter -> string -> unit
val pp_get_ellipsis_text : formatter -> unit -> string
val pp_set_formatter_out_channel :
  formatter -> Pervasives.out_channel -> unit
val pp_set_formatter_output_functions :
  formatter -> (string -> int -> int -> unit) -> (unit -> unit) -> unit
val pp_get_formatter_output_functions :
  formatter -> unit -> (string -> int -> int -> unit) * (unit -> unit)
val pp_set_formatter_tag_functions :
  formatter -> formatter_tag_functions -> unit
val pp_get_formatter_tag_functions :
  formatter -> unit -> formatter_tag_functions
val pp_set_formatter_out_functions :
  formatter -> formatter_out_functions -> unit
val pp_get_formatter_out_functions :
  formatter -> unit -> formatter_out_functions

```

These functions are the basic ones: usual functions operating on the standard formatter are defined via partial evaluation of these primitives. For instance, `print_string` is equal to `pp_print_string std_formatter`.

Convenience formatting functions.

```

val pp_print_list :
  ?pp_sep:(formatter -> unit -> unit) ->
  (formatter -> 'a -> unit) -> formatter -> 'a list -> unit

```

`pp_print_list ?pp_sep pp_v ppf l` prints the list `l`. `pp_v` is used on the elements of `l` and each element is separated by a call to `pp_sep` (defaults to `Format.pp_print_cut`[39]). Does nothing on empty lists.

Since: 4.02.0

```

val pp_print_text : formatter -> string -> unit
  pp_print_text ppf s prints s with spaces and newlines respectively printed with
  Format.pp_print_space[39] and Format.pp_force_newline[39].

```

Since: 4.02.0

`printf` like functions for pretty-printing.

```

val fprintf : formatter -> ('a, formatter, unit) Pervasives.format -> 'a
  fprintf ff fmt arg1 ... argN formats the arguments arg1 to argN according to the format
  string fmt, and outputs the resulting string on the formatter ff.

```

The format `fmt` is a character string which contains three types of objects: plain characters and conversion specifications as specified in the `Printf` module, and pretty-printing indications specific to the `Format` module.

The pretty-printing indication characters are introduced by a `@` character, and their meanings are:

- `@[`: open a pretty-printing box. The type and offset of the box may be optionally specified with the following syntax: the `<` character, followed by an optional box type indication, then an optional integer offset, and the closing `>` character. Box type is one of `h`, `v`, `hv`, `b`, or `hov`, which stand respectively for an horizontal box, a vertical box, an 'horizontal-vertical' box, or an 'horizontal or vertical' box (`b` standing for an 'horizontal or vertical' box demonstrating indentation and `hov` standing for a regular 'horizontal or vertical' box). For instance, `@[<hov 2>` opens an 'horizontal or vertical' box with indentation 2 as obtained with `open_hovbox 2`. For more details about boxes, see the various box opening functions `open_*box`.
- `@]`: close the most recently opened pretty-printing box.
- `@,`: output a good break hint, as with `print_cut ()`.
- `@ :` output a good break space, as with `print_space ()`.
- `@;`: output a fully specified good break as with `print_break`. The `nspaces` and `offset` parameters of the break may be optionally specified with the following syntax: the `<` character, followed by an integer `nspaces` value, then an integer `offset`, and a closing `>` character. If no parameters are provided, the good break defaults to a good break space.
- `@.`: flush the pretty printer and output a new line, as with `print_newline ()`.
- `@<n>`: print the following item as if it were of length `n`. Hence, `printf "@<0>%s" arg` prints `arg` as a zero length string. If `@<n>` is not followed by a conversion specification, then the following character of the format is printed as if it were of length `n`.
- `@{`: open a tag. The name of the tag may be optionally specified with the following syntax: the `<` character, followed by an optional string specification, and the closing `>` character. The string specification is any character string that does not contain the closing character `'>'`. If omitted, the tag name defaults to the empty string. For more details about tags, see the functions `open_tag` and `close_tag`.
- `@}`: close the most recently opened tag.
- `@?`: flush the pretty printer as with `print_flush ()`. This is equivalent to the conversion `%!`.
- `@\n`: force a newline, as with `force_newline ()`.
- `@@`: print a single `@` character.

Example: `printf "@[%s@ %d@]@." "x =" 1` is equivalent to `open_box (); print_string "x ="; print_space (); print_int 1; close_box (); print_newline ()`. It prints `x = 1` within a pretty-printing box.

Note: If you need to prevent the interpretation of a `@` character as a pretty-printing indication, you can also escape it with a `%` character.

```
val printf : ('a, formatter, unit) Pervasives.format -> 'a
```

Same as `fprintf` above, but output on `std_formatter`.

```
val eprintf : ('a, formatter, unit) Pervasives.format -> 'a
```

Same as `fprintf` above, but output on `err_formatter`.

val sprintf : ('a, unit, string) Pervasives.format -> 'a

Same as `printf` above, but instead of printing on a formatter, returns a string containing the result of formatting the arguments. Note that the pretty-printer queue is flushed at the end of *each call* to `sprintf`.

In case of multiple and related calls to `sprintf` to output material on a single string, you should consider using `fprintf` with the predefined formatter `str_formatter` and call `flush_str_formatter ()` to get the final result.

Alternatively, you can use `Format.fprintf` with a formatter writing to a buffer of your own: flushing the formatter and the buffer at the end of pretty-printing returns the desired string.

val asprintf : ('a, formatter, unit, string) Pervasives.format4 -> 'a

Same as `printf` above, but instead of printing on a formatter, returns a string containing the result of formatting the arguments. The type of `asprintf` is general enough to interact nicely with `%a` conversions.

Since: 4.01.0

val ifprintf : formatter -> ('a, formatter, unit) Pervasives.format -> 'a

Same as `fprintf` above, but does not print anything. Useful to ignore some material when conditionally printing.

Since: 3.10.0

Formatted output functions with continuations.

val kfprintf :

(formatter -> 'a) ->

formatter -> ('b, formatter, unit, 'a) Pervasives.format4 -> 'b

Same as `fprintf` above, but instead of returning immediately, passes the formatter to its first argument at the end of printing.

val ikfprintf :

(formatter -> 'a) ->

formatter -> ('b, formatter, unit, 'a) Pervasives.format4 -> 'b

Same as `kfprintf` above, but does not print anything. Useful to ignore some material when conditionally printing.

Since: 3.12.0

val ksprintf :

(string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b

Same as `sprintf` above, but instead of returning the string, passes it to the first argument.

Deprecated

val bprintf : Buffer.t -> ('a, formatter, unit) Pervasives.format -> 'a

Deprecated. This function is error prone. Do not use it.

If you need to print to some buffer `b`, you must first define a formatter writing to `b`, using `let to_b = formatter_of_buffer b`; then use regular calls to `Format.fprintf` on formatter `to_b`.

```
val kprintf :  
  (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b
```

Deprecated. An alias for `ksprintf`.

```
val set_all_formatter_output_functions :  
  out:(string -> int -> int -> unit) ->  
  flush:(unit -> unit) ->  
  newline:(unit -> unit) -> spaces:(int -> unit) -> unit
```

Deprecated. Subsumed by `set_formatter_out_functions`.

```
val get_all_formatter_output_functions :  
  unit ->  
  (string -> int -> int -> unit) * (unit -> unit) * (unit -> unit) *  
  (int -> unit)
```

Deprecated. Subsumed by `get_formatter_out_functions`.

```
val pp_set_all_formatter_output_functions :  
  formatter ->  
  out:(string -> int -> int -> unit) ->  
  flush:(unit -> unit) ->  
  newline:(unit -> unit) -> spaces:(int -> unit) -> unit
```

Deprecated. Subsumed by `pp_set_formatter_out_functions`.

```
val pp_get_all_formatter_output_functions :  
  formatter ->  
  unit ->  
  (string -> int -> int -> unit) * (unit -> unit) * (unit -> unit) *  
  (int -> unit)
```

Deprecated. Subsumed by `pp_get_formatter_out_functions`.

40 Module CamlinternalFormatBasics

```
type padty =  
  | Left  
  | Right  
  | Zeros
```

```

type int_conv =
  | Int_d
  | Int_pd
  | Int_sd
  | Int_i
  | Int_pi
  | Int_si
  | Int_x
  | Int_Cx
  | Int_X
  | Int_CX
  | Int_o
  | Int_Co
  | Int_u

type float_conv =
  | Float_f
  | Float_pf
  | Float_sf
  | Float_e
  | Float_pe
  | Float_se
  | Float_E
  | Float_pE
  | Float_sE
  | Float_g
  | Float_pg
  | Float_sg
  | Float_G
  | Float_pG
  | Float_sG
  | Float_F

type char_set = string

type counter =
  | Line_counter
  | Char_counter
  | Token_counter

type ('a, 'b) padding =
  | No_padding : ('a0, 'a0) padding
  | Lit_padding : padty * int -> ('a1, 'a1) padding
  | Arg_padding : padty -> (int -> 'a2, 'a2) padding

type pad_option = int option

type ('a, 'b) precision =
  | No_precision : ('a0, 'a0) precision
  | Lit_precision : int -> ('a1, 'a1) precision

```

```

    | Arg_precision : (int -> 'a2, 'a2) precision
type prec_option = int option
type ('a, 'b, 'c) custom_arity =
    | Custom_zero : ('a0, string, 'a0) custom_arity
    | Custom_succ : ('a1, 'b0, 'c0) custom_arity -> ('a1, 'x -> 'b0, 'x -> 'c0) custom_arity
type block_type =
    | Pp_hbox
    | Pp_vbox
    | Pp_hvbox
    | Pp_hovbox
    | Pp_box
    | Pp_fits
type formatting_lit =
    | Close_box
    | Close_tag
    | Break of string * int * int
    | FFlush
    | Force_newline
    | Flush_newline
    | Magic_size of string * int
    | Escaped_at
    | Escaped_percent
    | Scan_indic of char
type ('a, 'b, 'c, 'd, 'e, 'f) formatting_gen =
    | Open_tag : ('a0, 'b0, 'c0, 'd0, 'e0, 'f0) format6 -> ('a0, 'b0, 'c0, 'd0, 'e0, 'f0) format6
    | Open_box : ('a1, 'b1, 'c1, 'd1, 'e1, 'f1) format6 -> ('a1, 'b1, 'c1, 'd1, 'e1, 'f1) format6
type ('a, 'b, 'c, 'd, 'e, 'f) fmtty = ('a, 'b, 'c, 'd, 'e, 'f, 'a, 'b, 'c, 'd, 'e, 'f)
    fmtty_rel
type ('a1, 'b1, 'c1, 'd1, 'e1, 'f1, 'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel =
    | Char_ty : ('a10, 'b10, 'c10, 'd10, 'e10, 'f10, 'a20, 'b20, 'c20, 'd20, 'e20, 'f20)
    fmtty_rel -> (char -> 'a10, 'b10, 'c10, 'd10, 'e10, 'f10, char -> 'a20, 'b20, 'c20, 'd20,
        'e20, 'f20)
    fmtty_rel
    | String_ty : ('a11, 'b11, 'c11, 'd11, 'e11, 'f11, 'a21, 'b21, 'c21, 'd21, 'e21, 'f21)
    fmtty_rel -> (string -> 'a11, 'b11, 'c11, 'd11, 'e11, 'f11, string -> 'a21, 'b21, 'c21,
        'd21, 'e21, 'f21)
    fmtty_rel
    | Int_ty : ('a12, 'b12, 'c12, 'd12, 'e12, 'f12, 'a22, 'b22, 'c22, 'd22, 'e22, 'f22)
    fmtty_rel -> (int -> 'a12, 'b12, 'c12, 'd12, 'e12, 'f12, int -> 'a22, 'b22, 'c22, 'd22,
        'e22, 'f22)
    fmtty_rel
    | Int32_ty : ('a13, 'b13, 'c13, 'd13, 'e13, 'f13, 'a23, 'b23, 'c23, 'd23, 'e23, 'f23)
    fmtty_rel -> (int32 -> 'a13, 'b13, 'c13, 'd13, 'e13, 'f13, int32 -> 'a23, 'b23, 'c23,
        'd23, 'e23, 'f23)

```

```

fmtty_rel
| Nativeint_ty : ('a14, 'b14, 'c14, 'd14, 'e14, 'f14, 'a24, 'b24, 'c24, 'd24, 'e24, 'f24)
fmtty_rel -> (nativeint -> 'a14, 'b14, 'c14, 'd14, 'e14, 'f14, nativeint -> 'a24, 'b24,
'c24, 'd24, 'e24, 'f24)
fmtty_rel
| Int64_ty : ('a15, 'b15, 'c15, 'd15, 'e15, 'f15, 'a25, 'b25, 'c25, 'd25, 'e25, 'f25)
fmtty_rel -> (int64 -> 'a15, 'b15, 'c15, 'd15, 'e15, 'f15, int64 -> 'a25, 'b25, 'c25,
'd25, 'e25, 'f25)
fmtty_rel
| Float_ty : ('a16, 'b16, 'c16, 'd16, 'e16, 'f16, 'a26, 'b26, 'c26, 'd26, 'e26, 'f26)
fmtty_rel -> (float -> 'a16, 'b16, 'c16, 'd16, 'e16, 'f16, float -> 'a26, 'b26, 'c26,
'd26, 'e26, 'f26)
fmtty_rel
| Bool_ty : ('a17, 'b17, 'c17, 'd17, 'e17, 'f17, 'a27, 'b27, 'c27, 'd27, 'e27, 'f27)
fmtty_rel -> (bool -> 'a17, 'b17, 'c17, 'd17, 'e17, 'f17, bool -> 'a27, 'b27, 'c27, 'd27,
'e27, 'f27)
fmtty_rel
| Format_arg_ty : ('g, 'h, 'i, 'j, 'k, 'l) fmtty
* ('a18, 'b18, 'c18, 'd18, 'e18, 'f18, 'a28, 'b28, 'c28, 'd28, 'e28, 'f28)
fmtty_rel -> (('g, 'h, 'i, 'j, 'k, 'l) format6 -> 'a18, 'b18,
'c18, 'd18, 'e18, 'f18,
('g, 'h, 'i, 'j, 'k, 'l) format6 -> 'a28, 'b28,
'c28, 'd28, 'e28, 'f28)
fmtty_rel
| Format_subst_ty : ('g0, 'h0, 'i0, 'j0, 'k0, 'l0, 'g1, 'b19, 'c19, 'j1, 'd19, 'a19)
fmtty_rel
* ('g0, 'h0, 'i0, 'j0, 'k0, 'l0, 'g2, 'b29, 'c29, 'j2, 'd29, 'a29)
fmtty_rel
* ('a19, 'b19, 'c19, 'd19, 'e19, 'f19, 'a29, 'b29, 'c29, 'd29, 'e29, 'f29)
fmtty_rel -> (('g0, 'h0, 'i0, 'j0, 'k0, 'l0) format6 -> 'g1,
'b19, 'c19, 'j1, 'e19, 'f19,
('g0, 'h0, 'i0, 'j0, 'k0, 'l0) format6 -> 'g2,
'b29, 'c29, 'j2, 'e29, 'f29)
fmtty_rel
| Alpha_ty : ('a110, 'b110, 'c110, 'd110, 'e110, 'f110, 'a210, 'b210, 'c210, 'd210, 'e210,
'f210)
fmtty_rel -> (('b110 -> 'x -> 'c110) -> 'x -> 'a110, 'b110, 'c110, 'd110, 'e110, 'f110,
('b210 -> 'x -> 'c210) -> 'x -> 'a210, 'b210, 'c210, 'd210, 'e210, 'f210)
fmtty_rel
| Theta_ty : ('a111, 'b111, 'c111, 'd111, 'e111, 'f111, 'a211, 'b211, 'c211, 'd211, 'e211,
'f211)
fmtty_rel -> (('b111 -> 'c111) -> 'a111, 'b111, 'c111, 'd111, 'e111, 'f111,
('b211 -> 'c211) -> 'a211, 'b211, 'c211, 'd211, 'e211, 'f211)
fmtty_rel
| Any_ty : ('a112, 'b112, 'c112, 'd112, 'e112, 'f112, 'a212, 'b212, 'c212, 'd212, 'e212,

```

```

'f212)
fmtty_rel -> ('x0 -> 'a112, 'b112, 'c112, 'd112, 'e112, 'f112, 'x0 -> 'a212, 'b212, 'c212,
'd212, 'e212, 'f212)
fmtty_rel
| Reader_ty : ('a113, 'b113, 'c113, 'd113, 'e113, 'f113, 'a213, 'b213, 'c213, 'd213, 'e213,
'f213)
fmtty_rel -> ('x1 -> 'a113, 'b113, 'c113, ('b113 -> 'x1) -> 'd113, 'e113, 'f113,
'x1 -> 'a213, 'b213, 'c213, ('b213 -> 'x1) -> 'd213, 'e213, 'f213)
fmtty_rel
| Ignored_reader_ty : ('a114, 'b114, 'c114, 'd114, 'e114, 'f114, 'a214, 'b214, 'c214, 'd214,
'f214)
fmtty_rel -> ('a114, 'b114, 'c114, ('b114 -> 'x2) -> 'd114, 'e114, 'f114, 'a214, 'b214,
'c214, ('b214 -> 'x2) -> 'd214, 'e214, 'f214)
fmtty_rel
| End_of_fmtty : ('f115, 'b115, 'c115, 'd115, 'e115, 'f115, 'f215, 'b215, 'c215, 'd215, 'e215,
'f215)
fmtty_rel
type ('a, 'b, 'c, 'd, 'e, 'f) fmt =
| Char : ('a0, 'b0, 'c0, 'd0, 'e0, 'f0) fmt -> (char -> 'a0, 'b0, 'c0, 'd0, 'e0, 'f0) fmt
| Caml_char : ('a1, 'b1, 'c1, 'd1, 'e1, 'f1) fmt -> (char -> 'a1, 'b1, 'c1, 'd1, 'e1, 'f1)
| String : ('x, string -> 'a2) padding
* ('a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmt -> ('x, 'b2, 'c2, 'd2, 'e2, 'f2) fmt
| Caml_string : ('x0, string -> 'a3) padding
* ('a3, 'b3, 'c3, 'd3, 'e3, 'f3) fmt -> ('x0, 'b3, 'c3, 'd3, 'e3, 'f3) fmt
| Int : int_conv
* ('x1, 'y) padding
* ('y, int -> 'a4) precision
* ('a4, 'b4, 'c4, 'd4, 'e4, 'f4) fmt -> ('x1, 'b4, 'c4, 'd4, 'e4, 'f4) fmt
| Int32 : int_conv
* ('x2, 'y0) padding
* ('y0, int32 -> 'a5) precision
* ('a5, 'b5, 'c5, 'd5, 'e5, 'f5) fmt -> ('x2, 'b5, 'c5, 'd5, 'e5, 'f5) fmt
| Nativeint : int_conv
* ('x3, 'y1) padding
* ('y1, nativeint -> 'a6) precision
* ('a6, 'b6, 'c6, 'd6, 'e6, 'f6) fmt -> ('x3, 'b6, 'c6, 'd6, 'e6, 'f6) fmt
| Int64 : int_conv
* ('x4, 'y2) padding
* ('y2, int64 -> 'a7) precision
* ('a7, 'b7, 'c7, 'd7, 'e7, 'f7) fmt -> ('x4, 'b7, 'c7, 'd7, 'e7, 'f7) fmt
| Float : float_conv
* ('x5, 'y3) padding
* ('y3, float -> 'a8) precision
* ('a8, 'b8, 'c8, 'd8, 'e8, 'f8) fmt -> ('x5, 'b8, 'c8, 'd8, 'e8, 'f8) fmt
| Bool : ('a9, 'b9, 'c9, 'd9, 'e9, 'f9) fmt -> (bool -> 'a9, 'b9, 'c9, 'd9, 'e9, 'f9) fmt

```

```

| Flush : ('a10, 'b10, 'c10, 'd10, 'e10, 'f10) fmt -> ('a10, 'b10, 'c10, 'd10, 'e10, 'f10)
| String_literal : string * ('a11, 'b11, 'c11, 'd11, 'e11, 'f11) fmt -> ('a11, 'b11, 'c11,
| Char_literal : char * ('a12, 'b12, 'c12, 'd12, 'e12, 'f12) fmt -> ('a12, 'b12, 'c12, 'd12
| Format_arg : pad_option
* ('g, 'h, 'i, 'j, 'k, 'l) fmtty
* ('a13, 'b13, 'c13, 'd13, 'e13, 'f13) fmt -> (('g, 'h, 'i, 'j, 'k, 'l) format6 -> 'a13, '
'c13, 'd13, 'e13, 'f13)
fmt
| Format_subst : pad_option
* ('g0, 'h0, 'i0, 'j0, 'k0, 'l0, 'g2, 'b14, 'c14, 'j2, 'd14, 'a14)
fmtty_rel
* ('a14, 'b14, 'c14, 'd14, 'e14, 'f14) fmt -> (('g0, 'h0, 'i0, 'j0, 'k0, 'l0) format6 -> '
'b14, 'c14, 'j2, 'e14, 'f14)
fmt
| Alpha : ('a15, 'b15, 'c15, 'd15, 'e15, 'f15) fmt -> (('b15 -> 'x6 -> 'c15) -> 'x6 -> 'a15
fmt
| Theta : ('a16, 'b16, 'c16, 'd16, 'e16, 'f16) fmt -> (('b16 -> 'c16) -> 'a16, 'b16, 'c16,
fmt
| Formatting_lit : formatting_lit
* ('a17, 'b17, 'c17, 'd17, 'e17, 'f17) fmt -> ('a17, 'b17, 'c17, 'd17, 'e17, 'f17) fmt
| Formatting_gen : ('a18, 'b18, 'c18, 'd18, 'e18, 'f18) formatting_gen
* ('f18, 'b18, 'c18, 'e18, 'e20, 'f20) fmt -> ('a18, 'b18, 'c18, 'd18, 'e20, 'f20) fmt
| Reader : ('a19, 'b19, 'c19, 'd19, 'e19, 'f19) fmt -> ('x7 -> 'a19, 'b19, 'c19, ('b19 -> '
fmt
| Scan_char_set : pad_option * char_set
* ('a20, 'b20, 'c20, 'd20, 'e21, 'f21) fmt -> (string -> 'a20, 'b20, 'c20, 'd20, 'e21, 'f2
| Scan_get_counter : counter
* ('a21, 'b21, 'c21, 'd21, 'e22, 'f22) fmt -> (int -> 'a21, 'b21, 'c21, 'd21, 'e22, 'f22)
| Scan_next_char : ('a22, 'b22, 'c22, 'd22, 'e23, 'f23) fmt -> (char -> 'a22, 'b22, 'c22, '
| Ignored_param : ('a23, 'b23, 'c23, 'd23, 'y4, 'x8) ignored
* ('x8, 'b23, 'c23, 'y4, 'e24, 'f24) fmt -> ('a23, 'b23, 'c23, 'd23, 'e24, 'f24) fmt
| Custom : ('a24, 'x9, 'y5) custom_arity * (unit -> 'x9)
* ('a24, 'b24, 'c24, 'd24, 'e25, 'f25) fmt -> ('y5, 'b24, 'c24, 'd24, 'e25, 'f25) fmt
| End_of_format : ('f26, 'b25, 'c25, 'e26, 'e26, 'f26) fmt

```

List of format elements.

```

type ('a, 'b, 'c, 'd, 'e, 'f) ignored =
| Ignored_char : ('a0, 'b0, 'c0, 'd0, 'd0, 'a0) ignored
| Ignored_caml_char : ('a1, 'b1, 'c1, 'd1, 'd1, 'a1) ignored
| Ignored_string : pad_option -> ('a2, 'b2, 'c2, 'd2, 'd2, 'a2) ignored
| Ignored_caml_string : pad_option -> ('a3, 'b3, 'c3, 'd3, 'd3, 'a3) ignored
| Ignored_int : int_conv * pad_option -> ('a4, 'b4, 'c4, 'd4, 'd4, 'a4) ignored
| Ignored_int32 : int_conv * pad_option -> ('a5, 'b5, 'c5, 'd5, 'd5, 'a5) ignored
| Ignored_nativeint : int_conv * pad_option -> ('a6, 'b6, 'c6, 'd6, 'd6, 'a6) ignored
| Ignored_int64 : int_conv * pad_option -> ('a7, 'b7, 'c7, 'd7, 'd7, 'a7) ignored
| Ignored_float : pad_option * prec_option -> ('a8, 'b8, 'c8, 'd8, 'd8, 'a8) ignored

```

```

| Ignored_bool : ('a9, 'b9, 'c9, 'd9, 'd9, 'a9) ignored
| Ignored_format_arg : pad_option
  * ('g, 'h, 'i, 'j, 'k, 'l) fmttty -> ('a10, 'b10, 'c10, 'd10, 'd10, 'a10) ignored
| Ignored_format_subst : pad_option
  * ('a11, 'b11, 'c11, 'd11, 'e0, 'f0) fmttty -> ('a11, 'b11, 'c11, 'd11, 'e0, 'f0) ignored
| Ignored_reader : ('a12, 'b12, 'c12, ('b12 -> 'x) -> 'd12, 'd12, 'a12)
  ignored
| Ignored_scan_char_set : pad_option * char_set -> ('a13, 'b13, 'c13, 'd13, 'd13, 'a13) ign
| Ignored_scan_get_counter : counter -> ('a14, 'b14, 'c14, 'd14, 'd14, 'a14) ignored
| Ignored_scan_next_char : ('a15, 'b15, 'c15, 'd15, 'd15, 'a15) ignored
type ('a, 'b, 'c, 'd, 'e, 'f) format6 =
  | Format of ('a, 'b, 'c, 'd, 'e, 'f) fmt * string
val concat_fmttty :
  ('g1, 'b1, 'c1, 'j1, 'd1, 'a1, 'g2, 'b2, 'c2, 'j2, 'd2, 'a2)
  fmttty_rel ->
  ('a1, 'b1, 'c1, 'd1, 'e1, 'f1, 'a2, 'b2, 'c2, 'd2, 'e2, 'f2)
  fmttty_rel ->
  ('g1, 'b1, 'c1, 'j1, 'e1, 'f1, 'g2, 'b2, 'c2, 'j2, 'e2, 'f2)
  fmttty_rel
val erase_rel :
  ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k, 'l)
  fmttty_rel ->
  ('a, 'b, 'c, 'd, 'e, 'f) fmttty
val concat_fmt :
  ('a, 'b, 'c, 'd, 'e, 'f) fmt ->
  ('f, 'b, 'c, 'e, 'g, 'h) fmt ->
  ('a, 'b, 'c, 'd, 'g, 'h) fmt

```

41 Module Obj : Operations on internal representations of values.

Not for the casual user.

```

type t
val repr : 'a -> t
val obj : t -> 'a
val magic : 'a -> 'b
val is_block : t -> bool
val is_int : t -> bool
val tag : t -> int
val set_tag : t -> int -> unit
val size : t -> int
val field : t -> int -> t

```

```

val set_field : t -> int -> t -> unit
val double_field : t -> int -> float
val set_double_field : t -> int -> float -> unit
val new_block : int -> int -> t
val dup : t -> t
val truncate : t -> int -> unit
val add_offset : t -> Int32.t -> t
val first_non_constant_constructor_tag : int
val last_non_constant_constructor_tag : int
val lazy_tag : int
val closure_tag : int
val object_tag : int
val infix_tag : int
val forward_tag : int
val no_scan_tag : int
val abstract_tag : int
val string_tag : int
val double_tag : int
val double_array_tag : int
val custom_tag : int
val final_tag : int
val int_tag : int
val out_of_heap_tag : int
val unaligned_tag : int
val extension_name : 'a -> string
val extension_id : 'a -> int
val extension_slot : 'a -> t

```

The following two functions are deprecated. Use module `Marshal`[21] instead.

```

val marshal : t -> bytes
val unmarshal : bytes -> int -> t * int

```

42 Module Char : Character operations.

```

val code : char -> int

```

Return the ASCII code of the argument.

```

val chr : int -> char

```

Return the character with the given ASCII code. Raise `Invalid_argument "Char.chr"` if the argument is outside the range 0–255.

val escaped : char -> string

Return a string representing the given character, with special characters escaped following the lexical conventions of OCaml.

val lowercase : char -> char

Convert the given character to its equivalent lowercase character.

val uppercase : char -> char

Convert the given character to its equivalent uppercase character.

type t = char

An alias for the type of characters.

val compare : t -> t -> int

The comparison function for characters, with the same specification as `Pervasives.compare`[17]. Along with the type `t`, this function `compare` allows the module `Char` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

43 Module Stream : Streams and parsers.

type 'a t

The type of streams holding values of type `'a`.

exception Failure

Raised by parsers when none of the first components of the stream patterns is accepted.

exception Error of string

Raised by parsers when the first component of a stream pattern is accepted, but one of the following components is rejected.

Stream builders

val from : (int -> 'a option) -> 'a t

`Stream.from f` returns a stream built from the function `f`. To create a new stream element, the function `f` is called with the current stream count. The user function `f` must return either `Some <value>` for a value or `None` to specify the end of the stream.

Do note that the indices passed to `f` may not start at 0 in the general case. For example, [`< '0; '1; Stream.from f >`] would call `f` the first time with count 2.

val of_list : 'a list -> 'a t

Return the stream holding the elements of the list in the same order.

val of_string : string -> char t

Return the stream of the characters of the string parameter.

val of_bytes : bytes -> char t

Return the stream of the characters of the bytes parameter.

Since: 4.02.0

val of_channel : Pervasives.in_channel -> char t

Return the stream of the characters read from the input channel.

Stream iterator

val iter : ('a -> unit) -> 'a t -> unit

Stream.iter *f s* scans the whole stream *s*, applying function *f* in turn to each stream element encountered.

Predefined parsers

val next : 'a t -> 'a

Return the first element of the stream and remove it from the stream. Raise **Stream.Failure** if the stream is empty.

val empty : 'a t -> unit

Return () if the stream is empty, else raise **Stream.Failure**.

Useful functions

val peek : 'a t -> 'a option

Return **Some** of "the first element" of the stream, or **None** if the stream is empty.

val junk : 'a t -> unit

Remove the first element of the stream, possibly unfreezing it before.

val count : 'a t -> int

Return the current count of the stream elements, i.e. the number of the stream elements discarded.

val npeek : int -> 'a t -> 'a list

npeek *n* returns the list of the *n* first elements of the stream, or all its remaining elements if less than *n* elements are available.

44 Module String : String operations.

A string is an immutable data structure that contains a fixed-length sequence of (single-byte) characters. Each character can be accessed in constant time through its index.

Given a string `s` of length `l`, we can access each of the `l` characters of `s` via its index in the sequence. Indexes start at 0, and we will call an index valid in `s` if it falls within the range `[0..l-1]` (inclusive). A position is the point between two characters or at the beginning or end of the string. We call a position valid in `s` if it falls within the range `[0..l]` (inclusive). Note that the character at index `n` is between positions `n` and `n+1`.

Two parameters `start` and `len` are said to designate a valid substring of `s` if `len >= 0` and `start` and `start+len` are valid positions in `s`.

OCaml strings used to be modifiable in place, for instance via the `String.set`[44] and `String.blit`[44] functions described below. This usage is deprecated and only possible when the compiler is put in "unsafe-string" mode by giving the `-unsafe-string` command-line option (which is currently the default for reasons of backward compatibility). This is done by making the types `string` and `bytes` (see module `Bytes`[12]) interchangeable so that functions expecting byte sequences can also accept strings as arguments and modify them.

All new code should avoid this feature and be compiled with the `-safe-string` command-line option to enforce the separation between the types `string` and `bytes`.

```
val length : string -> int
```

Return the length (number of characters) of the given string.

```
val get : string -> int -> char
```

`String.get s n` returns the character at index `n` in string `s`. You can also write `s.[n]` instead of `String.get s n`.

Raise `Invalid_argument` if `n` not a valid index in `s`.

```
val set : bytes -> int -> char -> unit
```

Deprecated. This is a deprecated alias of `Bytes.set`[12]. `String.set s n c` modifies byte sequence `s` in place, replacing the byte at index `n` with `c`. You can also write `s.[n] <- c` instead of `String.set s n c`.

Raise `Invalid_argument` if `n` is not a valid index in `s`.

```
val create : int -> bytes
```

Deprecated. This is a deprecated alias of `Bytes.create`[12]. `String.create n` returns a fresh byte sequence of length `n`. The sequence is uninitialized and contains arbitrary bytes.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length`[22].

```
val make : int -> char -> string
```

`String.make n c` returns a fresh string of length `n`, filled with the character `c`.

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length`[22].

```
val init : int -> (int -> char) -> string
```

`String.init n f` returns a string of length `n`, with character `i` initialized to the result of `f i` (called in increasing index order).

Raise `Invalid_argument` if `n < 0` or `n > Sys.max_string_length[22]`.

Since: 4.02.0

`val copy : string -> string`

Deprecated. Because strings are immutable, it doesn't make much sense to make identical copies of them. Return a copy of the given string.

`val sub : string -> int -> int -> string`

`String.sub s start len` returns a fresh string of length `len`, containing the substring of `s` that starts at position `start` and has length `len`.

Raise `Invalid_argument` if `start` and `len` do not designate a valid substring of `s`.

`val fill : bytes -> int -> int -> char -> unit`

Deprecated. This is a deprecated alias of `Bytes.fill[12]`. `String.fill s start len c` modifies byte sequence `s` in place, replacing `len` bytes with `c`, starting at `start`.

Raise `Invalid_argument` if `start` and `len` do not designate a valid range of `s`.

`val blit : string -> int -> bytes -> int -> int -> unit`

Same as `Bytes.blit_string[12]`.

`val concat : string -> string list -> string`

`String.concat sep sl` concatenates the list of strings `sl`, inserting the separator string `sep` between each.

Raise `Invalid_argument` if the result is longer than `Sys.max_string_length[22]` bytes.

`val iter : (char -> unit) -> string -> unit`

`String.iter f s` applies function `f` in turn to all the characters of `s`. It is equivalent to `f s.[0]; f s.[1]; ...; f s.[String.length s - 1]; ()`.

`val iteri : (int -> char -> unit) -> string -> unit`

Same as `String.iter[44]`, but the function is applied to the index of the element as first argument (counting from 0), and the character itself as second argument.

Since: 4.00.0

`val map : (char -> char) -> string -> string`

`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.

Since: 4.00.0

`val mapi : (int -> char -> char) -> string -> string`

`String.mapi` `f s` calls `f` with each character of `s` and its index (in increasing index order) and stores the results in a new string that is returned.

Since: 4.02.0

`val trim : string -> string`

Return a copy of the argument, without leading and trailing whitespace. The characters regarded as whitespace are: ' ', '\012', '\n', '\r', and '\t'. If there is neither leading nor trailing whitespace character in the argument, return the original string itself, not a copy.

Since: 4.00.0

`val escaped : string -> string`

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of OCaml. If there is no special character in the argument, return the original string itself, not a copy. Its inverse function is `Scanf.unescaped`.

Raise `Invalid_argument` if the result is longer than `Sys.max_string_length[22]` bytes.

`val index : string -> char -> int`

`String.index s c` returns the index of the first occurrence of character `c` in string `s`.

Raise `Not_found` if `c` does not occur in `s`.

`val rindex : string -> char -> int`

`String.rindex s c` returns the index of the last occurrence of character `c` in string `s`.

Raise `Not_found` if `c` does not occur in `s`.

`val index_from : string -> int -> char -> int`

`String.index_from s i c` returns the index of the first occurrence of character `c` in string `s` after position `i`. `String.index s c` is equivalent to `String.index_from s 0 c`.

Raise `Invalid_argument` if `i` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` after position `i`.

`val rindex_from : string -> int -> char -> int`

`String.rindex_from s i c` returns the index of the last occurrence of character `c` in string `s` before position `i+1`. `String.rindex s c` is equivalent to `String.rindex_from s (String.length s - 1) c`.

Raise `Invalid_argument` if `i+1` is not a valid position in `s`. Raise `Not_found` if `c` does not occur in `s` before position `i+1`.

`val contains : string -> char -> bool`

`String.contains s c` tests if character `c` appears in the string `s`.

`val contains_from : string -> int -> char -> bool`

`String.contains_from s start c` tests if character `c` appears in `s` after position `start`.
`String.contains s c` is equivalent to `String.contains_from s 0 c`.
 Raise `Invalid_argument` if `start` is not a valid position in `s`.

`val rcontains_from : string -> int -> char -> bool`
`String.rcontains_from s stop c` tests if character `c` appears in `s` before position `stop+1`.
 Raise `Invalid_argument` if `stop < 0` or `stop+1` is not a valid position in `s`.

`val uppercase : string -> string`
 Return a copy of the argument, with all lowercase letters translated to uppercase, including accented letters of the ISO Latin-1 (8859-1) character set.

`val lowercase : string -> string`
 Return a copy of the argument, with all uppercase letters translated to lowercase, including accented letters of the ISO Latin-1 (8859-1) character set.

`val capitalize : string -> string`
 Return a copy of the argument, with the first character set to uppercase.

`val uncapitalize : string -> string`
 Return a copy of the argument, with the first character set to lowercase.

`type t = string`
 An alias for the type of strings.

`val compare : t -> t -> int`
 The comparison function for strings, with the same specification as `Pervasives.compare`[17]. Along with the type `t`, this function `compare` allows the module `String` to be passed as argument to the functors `Set.Make`[38] and `Map.Make`[3].

45 Module Filename : Operations on file names.

`val current_dir_name : string`
 The conventional name for the current directory (e.g. `.` in Unix).

`val parent_dir_name : string`
 The conventional name for the parent of the current directory (e.g. `..` in Unix).

`val dir_sep : string`
 The directory separator (e.g. `/` in Unix).
Since: 3.11.2

```

val concat : string -> string -> string
    concat dir file returns a file name that designates file file in directory dir.

val is_relative : string -> bool
    Return true if the file name is relative to the current directory, false if it is absolute (i.e. in
    Unix, starts with /).

val is_implicit : string -> bool
    Return true if the file name is relative and does not start with an explicit reference to the
    current directory (./ or ../ in Unix), false if it starts with an explicit reference to the root
    directory or the current directory.

val check_suffix : string -> string -> bool
    check_suffix name suff returns true if the filename name ends with the suffix suff.

val chop_suffix : string -> string -> string
    chop_suffix name suff removes the suffix suff from the filename name. The behavior is
    undefined if name does not end with the suffix suff.

val chop_extension : string -> string
    Return the given file name without its extension. The extension is the shortest suffix
    starting with a period and not including a directory separator, .xyz for instance.
    Raise Invalid_argument if the given name does not contain an extension.

val basename : string -> string
    Split a file name into directory name / base file name. If name is a valid file name, then
    concat (dirname name) (basename name) returns a file name which is equivalent to name.
    Moreover, after setting the current directory to dirname name (with Sys.chdir[22]),
    references to basename name (which is a relative file name) designate the same file as name
    before the call to Sys.chdir[22].
    This function conforms to the specification of POSIX.1-2008 for the basename utility.

val dirname : string -> string
    See Filename.basename[45]. This function conforms to the specification of POSIX.1-2008
    for the dirname utility.

val temp_file : ?temp_dir:string -> string -> string -> string
    temp_file prefix suffix returns the name of a fresh temporary file in the temporary
    directory. The base name of the temporary file is formed by concatenating prefix, then a
    suitably chosen integer number, then suffix. The optional argument temp_dir indicates the
    temporary directory to use, defaulting to the current result of
    Filename.get_temp_dir_name[45]. The temporary file is created empty, with permissions
    0o600 (readable and writable only by the file owner). The file is guaranteed to be different
    from any other file that existed when temp_file was called. Raise Sys_error if the file
    could not be created.

Before 3.11.2 no ?temp_dir optional argument

```

val `open_temp_file` :

`?mode:Pervasives.open_flag list ->`

`?temp_dir:string -> string -> string -> string * Pervasives.out_channel`

Same as `Filename.temp_file`[45], but returns both the name of a fresh temporary file, and an output channel opened (atomically) on this file. This function is more secure than `temp_file`: there is no risk that the temporary file will be modified (e.g. replaced by a symbolic link) before the program opens it. The optional argument `mode` is a list of additional flags to control the opening of the file. It can contain one or several of `Open_append`, `Open_binary`, and `Open_text`. The default is `[Open_text]` (open in text mode). Raise `Sys_error` if the file could not be opened.

Before 3.11.2 no `?temp_dir` optional argument

val `get_temp_dir_name` : `unit -> string`

The name of the temporary directory: Under Unix, the value of the `TMPDIR` environment variable, or `"/tmp"` if the variable is not set. Under Windows, the value of the `TEMP` environment variable, or `""` if the variable is not set. The temporary directory can be changed with `Filename.set_temp_dir_name`[45].

Since: 4.00.0

val `set_temp_dir_name` : `string -> unit`

Change the temporary directory returned by `Filename.get_temp_dir_name`[45] and used by `Filename.temp_file`[45] and `Filename.open_temp_file`[45].

Since: 4.00.0

val `temp_dir_name` : `string`

Deprecated. You should use `Filename.get_temp_dir_name`[45] instead. The name of the initial temporary directory: Under Unix, the value of the `TMPDIR` environment variable, or `"/tmp"` if the variable is not set. Under Windows, the value of the `TEMP` environment variable, or `""` if the variable is not set.

Since: 3.09.1

val `quote` : `string -> string`

Return a quoted version of a file name, suitable for use as one argument in a command line, escaping all meta-characters. Warning: under Windows, the output is only suitable for use with programs that follow the standard Windows quoting conventions.

46 Module `Printexc` : Facilities for printing exceptions and inspecting current call stack.

val `to_string` : `exn -> string`

`Printexc.to_string e` returns a string representation of the exception `e`.

val print : ('a -> 'b) -> 'a -> 'b

`Printexc.print fn x` applies `fn` to `x` and returns the result. If the evaluation of `fn x` raises any exception, the name of the exception is printed on standard error output, and the exception is raised again. The typical use is to catch and report exceptions that escape a function application.

val catch : ('a -> 'b) -> 'a -> 'b

`Printexc.catch fn x` is similar to `Printexc.print`^[46], but aborts the program with exit code 2 after printing the uncaught exception. This function is deprecated: the runtime system is now able to print uncaught exceptions as precisely as `Printexc.catch` does. Moreover, calling `Printexc.catch` makes it harder to track the location of the exception using the debugger or the stack backtrace facility. So, do not use `Printexc.catch` in new code.

val print_backtrace : Pervasives.out_channel -> unit

`Printexc.print_backtrace oc` prints an exception backtrace on the output channel `oc`. The backtrace lists the program locations where the most-recently raised exception was raised and where it was propagated through function calls.

Since: 3.11.0

val get_backtrace : unit -> string

`Printexc.get_backtrace ()` returns a string containing the same exception backtrace that `Printexc.print_backtrace` would print.

Since: 3.11.0

val record_backtrace : bool -> unit

`Printexc.record_backtrace b` turns recording of exception backtraces on (if `b = true`) or off (if `b = false`). Initially, backtraces are not recorded, unless the `b` flag is given to the program through the `OCAMLRUNPARAM` variable.

Since: 3.11.0

val backtrace_status : unit -> bool

`Printexc.backtrace_status()` returns `true` if exception backtraces are currently recorded, `false` if not.

Since: 3.11.0

val register_printer : (exn -> string option) -> unit

`Printexc.register_printer fn` registers `fn` as an exception printer. The printer should return `None` or raise an exception if it does not know how to convert the passed exception, and `Some s` with `s` the resulting string if it can convert the passed exception. Exceptions raised by the printer are ignored.

When converting an exception into a string, the printers will be invoked in the reverse order of their registrations, until a printer returns a `Some s` value (if no such printer exists, the runtime will use a generic printer).

When using this mechanism, one should be aware that an exception backtrace is attached to the thread that saw it raised, rather than to the exception itself. Practically, it means that the code related to `fn` should not use the backtrace if it has itself raised an exception before.

Since: 3.11.2

Raw backtraces

type `raw_backtrace`

The abstract type `raw_backtrace` stores a backtrace in a low-level format, instead of directly exposing them as string as the `get_backtrace()` function does.

This allows delaying the formatting of backtraces to when they are actually printed, which may be useful if you record more backtraces than you print.

Raw backtraces cannot be marshalled. If you need marshalling, you should use the array returned by the `backtrace_slots` function of the next section.

Since: 4.01.0

val `get_raw_backtrace` : `unit` -> `raw_backtrace`

`Printexc.get_raw_backtrace ()` returns the same exception backtrace that `Printexc.print_backtrace` would print, but in a raw format.

Since: 4.01.0

val `print_raw_backtrace` : `Pervasives.out_channel` -> `raw_backtrace` -> `unit`

Print a raw backtrace in the same format `Printexc.print_backtrace` uses.

Since: 4.01.0

val `raw_backtrace_to_string` : `raw_backtrace` -> `string`

Return a string from a raw backtrace, in the same format `Printexc.get_backtrace` uses.

Since: 4.01.0

Current call stack

val `get_callstack` : `int` -> `raw_backtrace`

`Printexc.get_callstack n` returns a description of the top of the call stack on the current program point (for the current thread), with at most `n` entries. (Note: this function is not related to exceptions at all, despite being part of the `Printexc` module.)

Since: 4.01.0

Uncaught exceptions

val `set_uncaught_exception_handler` : (`exn` -> `raw_backtrace` -> `unit`) -> `unit`

`Printexc.set_uncaught_exception_handler fn` registers `fn` as the handler for uncaught exceptions. The default handler prints the exception and backtrace on standard error output.

Note that when `fn` is called all the functions registered with `Pervasives.at_exit[17]` have already been called. Because of this you must make sure any output channel `fn` writes on is flushed.

Also note that exceptions raised by user code in the interactive toplevel are not passed to this function as they are caught by the toplevel itself.

If `fn` raises an exception, both the exceptions passed to `fn` and raised by `fn` will be printed with their respective backtrace.

Since: 4.02.0

Manipulation of backtrace information

Those function allow to traverse the slots of a raw backtrace, extract information from them in a programmer-friendly format.

`type backtrace_slot`

The abstract type `backtrace_slot` represents a single slot of a backtrace.

Since: 4.02

`val backtrace_slots : raw_backtrace -> backtrace_slot array option`

Returns the slots of a raw backtrace, or `None` if none of them contain useful information.

In the return array, the slot at index 0 corresponds to the most recent function call, raise, or primitive `get_backtrace` call in the trace.

Some possible reasons for returning `None` are as follow:

- none of the slots in the trace come from modules compiled with debug information (`-g`)
- the program is a bytecode program that has not been linked with debug information enabled (`ocamlc -g`)

Since: 4.02.0

```
type location = {  
  filename : string ;  
  line_number : int ;  
  start_char : int ;  
  end_char : int ;  
}
```

The type of location information found in backtraces. `start_char` and `end_char` are positions relative to the beginning of the line.

Since: 4.02

`module Slot :`

`sig`

`type t = Printexc.backtrace_slot`

`val is_raise : t -> bool`

`is_raise slot` is `true` when `slot` refers to a raising point in the code, and `false` when it comes from a simple function call.

Since: 4.02

```
val location : t -> Printexc.location option
```

`location slot` returns the location information of the slot, if available, and `None` otherwise.

Some possible reasons for failing to return a location are as follow:

- the slot corresponds to a compiler-inserted raise
- the slot corresponds to a part of the program that has not been compiled with debug information (`-g`)

Since: 4.02

```
val format : int -> t -> string option
```

`format pos slot` returns the string representation of `slot` as `raw_backtrace_to_string` would format it, assuming it is the `pos`-th element of the backtrace: the 0-th element is pretty-printed differently than the others.

Whole-backtrace printing functions also skip some uninformative slots; in that case, `format pos slot` returns `None`.

Since: 4.02

end

Raw backtrace slots

```
type raw_backtrace_slot
```

This type allows direct access to raw backtrace slots, without any conversion in an OCaml-usable data-structure. Being process-specific, they must absolutely not be marshalled, and are unsafe to use for this reason (marshalling them may not fail, but un-marshalling and using the result will result in undefined behavior).

Elements of this type can still be compared and hashed: when two elements are equal, then they represent the same source location (the converse is not necessarily true in presence of inlining, for example).

Since: 4.02.0

```
val raw_backtrace_length : raw_backtrace -> int
```

`raw_backtrace_length bckt` returns the number of slots in the backtrace `bckt`.

Since: 4.02

```
val get_raw_backtrace_slot : raw_backtrace -> int -> raw_backtrace_slot
```

`get_slot bckt pos` returns the slot in position `pos` in the backtrace `bckt`.

Since: 4.02

```
val convert_raw_backtrace_slot : raw_backtrace_slot -> backtrace_slot
```

Extracts the user-friendly `backtrace_slot` from a low-level `raw_backtrace_slot`.

Since: 4.02

Exception slots

`val exn_slot_id : exn -> int`

`Printexc.exn_slot_id` returns an integer which uniquely identifies the constructor used to create the exception value `exn` (in the current runtime).

Since: 4.02.0

`val exn_slot_name : exn -> string`

`Printexc.exn_slot_id exn` returns the internal name of the constructor used to create the exception value `exn`.

Since: 4.02.0

47 Module Random : Pseudo-random number generators (PRNG).

Basic functions

`val init : int -> unit`

Initialize the generator, using the argument as a seed. The same seed will always yield the same sequence of numbers.

`val full_init : int array -> unit`

Same as `Random.init[47]` but takes more data as seed.

`val self_init : unit -> unit`

Initialize the generator with a random seed chosen in a system-dependent way. If `/dev/urandom` is available on the host machine, it is used to provide a highly random initial seed. Otherwise, a less random seed is computed from system parameters (current time, process IDs).

`val bits : unit -> int`

Return 30 random bits in a nonnegative integer.

Before 3.12.0 used a different algorithm (affects all the following functions)

`val int : int -> int`

`Random.int bound` returns a random integer between 0 (inclusive) and `bound` (exclusive). `bound` must be greater than 0 and less than 2^{30} .

`val int32 : Int32.t -> Int32.t`

`Random.int32 bound` returns a random integer between 0 (inclusive) and `bound` (exclusive). `bound` must be greater than 0.

`val nativeint : Nativeint.t -> Nativeint.t`

`Random.nativeint bound` returns a random integer between 0 (inclusive) and `bound` (exclusive). `bound` must be greater than 0.

```
val int64 : Int64.t -> Int64.t
```

`Random.int64 bound` returns a random integer between 0 (inclusive) and `bound` (exclusive). `bound` must be greater than 0.

```
val float : float -> float
```

`Random.float bound` returns a random floating-point number between 0 and `bound` (inclusive). If `bound` is negative, the result is negative or zero. If `bound` is 0, the result is 0.

```
val bool : unit -> bool
```

`Random.bool ()` returns `true` or `false` with probability 0.5 each.

Advanced functions

Advanced functions

The functions from module `State` manipulate the current state of the random generator explicitly. This allows using one or several deterministic PRNGs, even in a multi-threaded program, without interference from other parts of the program.

```
module State :
```

```
sig
```

```
  type t
```

The type of PRNG states.

```
  val make : int array -> t
```

Create a new state and initialize it with the given seed.

```
  val make_self_init : unit -> t
```

Create a new state and initialize it with a system-dependent low-entropy seed.

```
  val copy : t -> t
```

Return a copy of the given state.

```
  val bits : t -> int
```

```
  val int : t -> int -> int
```

```
  val int32 : t -> Int32.t -> Int32.t
```

```
  val nativeint : t -> Nativeint.t -> Nativeint.t
```

```
  val int64 : t -> Int64.t -> Int64.t
```

```
  val float : t -> float -> float
```

```
  val bool : t -> bool
```

These functions are the same as the basic functions, except that they use (and update) the given PRNG state instead of the default one.

```
end
```

```
val get_state : unit -> State.t
```

Return the current state of the generator used by the basic functions.

```
val set_state : State.t -> unit
```

Set the state of the generator used by the basic functions.

48 Module Weak : Arrays of weak pointers and hash tables of weak pointers.

Low-level functions

```
type 'a t
```

The type of arrays of weak pointers (weak arrays). A weak pointer is a value that the garbage collector may erase whenever the value is not used any more (through normal pointers) by the program. Note that finalisation functions are run after the weak pointers are erased.

A weak pointer is said to be full if it points to a value, empty if the value was erased by the GC.

Notes:

- Integers are not allocated and cannot be stored in weak arrays.
- Weak arrays cannot be marshaled using `Pervasives.output_value[17]` nor the functions of the `Marshal[21]` module.

```
val create : int -> 'a t
```

`Weak.create n` returns a new weak array of length `n`. All the pointers are initially empty. Raise `Invalid_argument` if `n` is negative or greater than `Sys.max_array_length[22]-1`.

```
val length : 'a t -> int
```

`Weak.length ar` returns the length (number of elements) of `ar`.

```
val set : 'a t -> int -> 'a option -> unit
```

`Weak.set ar n (Some el)` sets the `n`th cell of `ar` to be a (full) pointer to `el`; `Weak.set ar n None` sets the `n`th cell of `ar` to empty. Raise `Invalid_argument "Weak.set"` if `n` is not in the range 0 to `Weak.length[48] a - 1`.

```
val get : 'a t -> int -> 'a option
```

`Weak.get ar n` returns `None` if the `n`th cell of `ar` is empty, `Some x` (where `x` is the value) if it is full. Raise `Invalid_argument "Weak.get"` if `n` is not in the range 0 to `Weak.length[48] a - 1`.

```
val get_copy : 'a t -> int -> 'a option
```

`Weak.get_copy ar n` returns `None` if the `n`th cell of `ar` is empty, `Some x` (where `x` is a (shallow) copy of the value) if it is full. In addition to pitfalls with mutable values, the interesting difference with `get` is that `get_copy` does not prevent the incremental GC from erasing the value in its current cycle (`get` may delay the erasure to the next GC cycle). Raise `Invalid_argument "Weak.get"` if `n` is not in the range 0 to `Weak.length[48] a - 1`.

```
val check : 'a t -> int -> bool
```

`Weak.check ar n` returns `true` if the `n`th cell of `ar` is full, `false` if it is empty. Note that even if `Weak.check ar n` returns `true`, a subsequent `Weak.get[48] ar n` can return `None`.

```
val fill : 'a t -> int -> int -> 'a option -> unit
```

`Weak.fill ar ofs len el` sets to `el` all pointers of `ar` from `ofs` to `ofs + len - 1`. Raise `Invalid_argument "Weak.fill"` if `ofs` and `len` do not designate a valid subarray of `a`.

```
val blit : 'a t -> int -> 'a t -> int -> int -> unit
```

`Weak.blit ar1 off1 ar2 off2 len` copies `len` weak pointers from `ar1` (starting at `off1`) to `ar2` (starting at `off2`). It works correctly even if `ar1` and `ar2` are the same. Raise `Invalid_argument "Weak.blit"` if `off1` and `len` do not designate a valid subarray of `ar1`, or if `off2` and `len` do not designate a valid subarray of `ar2`.

Weak hash tables

Weak hash tables

A weak hash table is a hashed set of values. Each value may magically disappear from the set when it is not used by the rest of the program any more. This is normally used to share data structures without inducing memory leaks. Weak hash tables are defined on values from a `Hashtbl.Hashtype[27]` module; the `equal` relation and `hash` function are taken from that module. We will say that `v` is an instance of `x` if `equal x v` is `true`.

The `equal` relation must be able to work on a shallow copy of the values and give the same result as with the values themselves.

```
module type S =
```

```
sig
```

```
  type data
```

The type of the elements stored in the table.

```
  type t
```

The type of tables that contain elements of type `data`. Note that weak hash tables cannot be marshaled using `Pervasives.output_value[17]` or the functions of the `Marshal[21]` module.

```
  val create : int -> t
```

`create n` creates a new empty weak hash table, of initial size `n`. The table will grow as needed.

```
  val clear : t -> unit
```

Remove all elements from the table.

```
val merge : t -> data -> data
```

`merge t x` returns an instance of `x` found in `t` if any, or else adds `x` to `t` and return `x`.

```
val add : t -> data -> unit
```

`add t x` adds `x` to `t`. If there is already an instance of `x` in `t`, it is unspecified which one will be returned by subsequent calls to `find` and `merge`.

```
val remove : t -> data -> unit
```

`remove t x` removes from `t` one instance of `x`. Does nothing if there is no instance of `x` in `t`.

```
val find : t -> data -> data
```

`find t x` returns an instance of `x` found in `t`. Raise `Not_found` if there is no such element.

```
val find_all : t -> data -> data list
```

`find_all t x` returns a list of all the instances of `x` found in `t`.

```
val mem : t -> data -> bool
```

`mem t x` returns `true` if there is at least one instance of `x` in `t`, false otherwise.

```
val iter : (data -> unit) -> t -> unit
```

`iter f t` calls `f` on each element of `t`, in some unspecified order. It is not specified what happens if `f` tries to change `t` itself.

```
val fold : (data -> 'a -> 'a) -> t -> 'a -> 'a
```

`fold f t init` computes `(f d1 (... (f dN init)))` where `d1 ... dN` are the elements of `t` in some unspecified order. It is not specified what happens if `f` tries to change `t` itself.

```
val count : t -> int
```

Count the number of elements in the table. `count t` gives the same result as `fold (fun _ n -> n+1) t 0` but does not delay the deallocation of the dead elements.

```
val stats : t -> int * int * int * int * int * int
```

Return statistics on the table. The numbers are, in order: table length, number of entries, sum of bucket lengths, smallest bucket length, median bucket length, biggest bucket length.

end

The output signature of the functor `Weak.Make`[48].

```
module Make :  
  functor (H : Hashtbl.HashedType) -> S with type data = H.t  
  Functor building an implementation of the weak hash table structure.
```

49 Module Unix : Interface to the Unix system.

Note: all the functions of this module (except `error_message` and `handle_unix_error`) are liable to raise the `Unix_error` exception whenever the underlying system call signals an error.

Error report

```
type error =  
  | E2BIG  
      Argument list too long  
  | EACCES  
      Permission denied  
  | EAGAIN  
      Resource temporarily unavailable; try again  
  | EBADF  
      Bad file descriptor  
  | EBUSY  
      Resource unavailable  
  | ECHILD  
      No child process  
  | EDEADLK  
      Resource deadlock would occur  
  | EDOM  
      Domain error for math functions, etc.  
  | EEXIST  
      File exists  
  | EFAULT  
      Bad address  
  | EFBIG  
      File too large  
  | EINTR
```

	Function interrupted by signal
EINVAL	Invalid argument
EIO	Hardware I/O error
EISDIR	Is a directory
EMFILE	Too many open files by the process
EMLINK	Too many links
ENAMETOOLONG	Filename too long
ENFILE	Too many open files in the system
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Not an executable file
ENOLCK	No locks available
ENOMEM	Not enough memory
ENOSPC	No space left on device
ENOSYS	Function not supported
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Inappropriate I/O control operation

ENXIO	No such device or address
EPERM	Operation not permitted
EPIPE	Broken pipe
ERANGE	Result too large
EROFS	Read-only file system
ESPIPE	Invalid seek e.g. on a pipe
ESRCH	No such process
EXDEV	Invalid link
EWOULDBLOCK	Operation would block
EINPROGRESS	Operation now in progress
EALREADY	Operation already in progress
ENOTSOCK	Socket operation on non-socket
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
EPROTOTYPE	Protocol wrong type for socket
ENOPROTOOPT	Protocol not available
EPROTONOSUPPORT	Protocol not supported
ESOCKTNOSUPPORT	

	Socket type not supported
EOPNOTSUPP	Operation not supported on socket
EPFNOSUPPORT	Protocol family not supported
EAFNOSUPPORT	Address family not supported by protocol family
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Can't assign requested address
ENETDOWN	Network is down
ENETUNREACH	Network is unreachable
ENETRESET	Network dropped connection on reset
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by peer
ENOBUFS	No buffer space available
EISCONN	Socket is already connected
ENOTCONN	Socket is not connected
ESHUTDOWN	Can't send after socket shutdown
ETOOMANYREFS	Too many references: can't splice
ETIMEDOUT	Connection timed out
ECONNREFUSED	Connection refused

| EHOSTDOWN

Host is down

| EHOSTUNREACH

No route to host

| ELOOP

Too many levels of symbolic links

| EOVERFLOW

File size or position not representable

| EUNKNOWNERR of int

Unknown error

The type of error codes. Errors defined in the POSIX standard and additional errors from UNIX98 and BSD. All other errors are mapped to EUNKNOWNERR.

exception Unix_error of error * string * string

Raised by the system calls below when an error is encountered. The first component is the error code; the second component is the function name; the third component is the string parameter to the function, if it has one, or the empty string otherwise.

val error_message : error -> string

Return a string describing the given error code.

val handle_unix_error : ('a -> 'b) -> 'a -> 'b

handle_unix_error f x applies **f** to **x** and returns the result. If the exception **Unix_error** is raised, it prints a message describing the error and exits with code 2.

Access to the process environment

val environment : unit -> string array

Return the process environment, as an array of strings with the format “variable=value”.

val getenv : string -> string

Return the value associated to a variable in the process environment. Raise **Not_found** if the variable is unbound. (This function is identical to **Sys.getenv**[22].)

val putenv : string -> string -> unit

Unix.putenv name value sets the value associated to a variable in the process environment. **name** is the name of the environment variable, and **value** its new associated value.

Process handling

type process_status =

| WEXITED of int

The process terminated normally by **exit**; the argument is the return code.

| WSIGNALED of int

The process was killed by a signal; the argument is the signal number.

| WSTOPPED of int

The process was stopped by a signal; the argument is the signal number.

The termination status of a process. See module `Sys`[22] for the definitions of the standard signal numbers. Note that they are not the numbers used by the OS.

type wait_flag =

| WNOHANG

do not block if no child has died yet, but immediately return with a pid equal to 0.

| WUNTRACED

report also the children that receive stop signals.

Flags for `Unix.waitpid`[49].

val execv : string -> string array -> 'a

`execv prog args` execute the program in file `prog`, with the arguments `args`, and the current process environment. These `execv*` functions never return: on success, the current program is replaced by the new one; on failure, a `Unix.Unix_error`[49] exception is raised.

val execve : string -> string array -> string array -> 'a

Same as `Unix.execv`[49], except that the third argument provides the environment to the program executed.

val execvp : string -> string array -> 'a

Same as `Unix.execv`[49], except that the program is searched in the path.

val execvpe : string -> string array -> string array -> 'a

Same as `Unix.execve`[49], except that the program is searched in the path.

val fork : unit -> int

Fork a new process. The returned integer is 0 for the child process, the pid of the child process for the parent process.

val wait : unit -> int * process_status

Wait until one of the children processes die, and return its pid and termination status.

val waitpid : wait_flag list -> int -> int * process_status

Same as `Unix.wait`[49], but waits for the child process whose pid is given. A pid of -1 means wait for any child. A pid of 0 means wait for any child in the same process group as the current process. Negative pid arguments represent process groups. The list of options indicates whether `waitpid` should return immediately without waiting, and whether it should report stopped children.

val system : string -> process_status

Execute the given command, wait until it terminates, and return its termination status. The string is interpreted by the shell `/bin/sh` and therefore can contain redirections, quotes, variables, etc. The result `WEXITED 127` indicates that the shell couldn't be executed.

`val getpid : unit -> int`

Return the pid of the process.

`val getppid : unit -> int`

Return the pid of the parent process.

`val nice : int -> int`

Change the process priority. The integer argument is added to the “nice” value. (Higher values of the “nice” value mean lower priorities.) Return the new nice value.

Basic file input/output

`type file_descr`

The abstract type of file descriptors.

`val stdin : file_descr`

File descriptor for standard input.

`val stdout : file_descr`

File descriptor for standard output.

`val stderr : file_descr`

File descriptor for standard error.

`type open_flag =`

| `O_RDONLY`

Open for reading

| `O_WRONLY`

Open for writing

| `O_RDWR`

Open for reading and writing

| `O_NONBLOCK`

Open in non-blocking mode

| `O_APPEND`

Open for append

| `O_CREAT`

Create if nonexistent

| `O_TRUNC`

```

        Truncate to 0 length if existing
| O_EXCL
        Fail if existing
| O_NOCTTY
        Don't make this dev a controlling tty
| O_DSYNC
        Writes complete as 'Synchronised I/O data integrity completion'
| O_SYNC
        Writes complete as 'Synchronised I/O file integrity completion'
| O_RSYNC
        Reads complete as writes (depending on O_SYNC/O_DSYNC)
| O_SHARE_DELETE
        Windows only: allow the file to be deleted while still open
| O_CLOEXEC
        Set the close-on-exec flag on the descriptor returned by Unix.openfile[49]
        The flags to Unix.openfile[49].

type file_perm = int
    The type of file access rights, e.g. 0o640 is read and write for user, read for group, none for others

val openfile : string -> open_flag list -> file_perm -> file_descr
    Open the named file with the given flags. Third argument is the permissions to give to the file if it is created (see Unix.umask[49]). Return a file descriptor on the named file.

val close : file_descr -> unit
    Close a file descriptor.

val read : file_descr -> bytes -> int -> int -> int
    read fd buff ofs len reads len bytes from descriptor fd, storing them in byte sequence buff, starting at position ofs in buff. Return the number of bytes actually read.

val write : file_descr -> bytes -> int -> int -> int
    write fd buff ofs len writes len bytes to descriptor fd, taking them from byte sequence buff, starting at position ofs in buff. Return the number of bytes actually written. write repeats the writing operation until all bytes have been written or an error occurs.

val single_write : file_descr -> bytes -> int -> int -> int
    Same as write, but attempts to write only once. Thus, if an error occurs, single_write guarantees that no data has been written.

```

```
val write_substring : file_descr -> string -> int -> int -> int
```

Same as `write`, but take the data from a string instead of a byte sequence.

```
val single_write_substring : file_descr -> string -> int -> int -> int
```

Same as `single_write`, but take the data from a string instead of a byte sequence.

Interfacing with the standard input/output library

```
val in_channel_of_descr : file_descr -> Pervasives.in_channel
```

Create an input channel reading from the given descriptor. The channel is initially in binary mode; use `set_binary_mode_in ic false` if text mode is desired.

```
val out_channel_of_descr : file_descr -> Pervasives.out_channel
```

Create an output channel writing on the given descriptor. The channel is initially in binary mode; use `set_binary_mode_out oc false` if text mode is desired.

```
val descr_of_in_channel : Pervasives.in_channel -> file_descr
```

Return the descriptor corresponding to an input channel.

```
val descr_of_out_channel : Pervasives.out_channel -> file_descr
```

Return the descriptor corresponding to an output channel.

Seeking and truncating

```
type seek_command =
```

```
| SEEK_SET
```

indicates positions relative to the beginning of the file

```
| SEEK_CUR
```

indicates positions relative to the current position

```
| SEEK_END
```

indicates positions relative to the end of the file

Positioning modes for `Unix.lseek`[49].

```
val lseek : file_descr -> int -> seek_command -> int
```

Set the current position for a file descriptor, and return the resulting offset (from the beginning of the file).

```
val truncate : string -> int -> unit
```

Truncates the named file to the given size.

```
val ftruncate : file_descr -> int -> unit
```

Truncates the file corresponding to the given descriptor to the given size.

File status

```
type file_kind =
```

```
| S_REG
```

```

        Regular file
| S_DIR
        Directory
| S_CHR
        Character device
| S_BLK
        Block device
| S_LNK
        Symbolic link
| S_FIFO
        Named pipe
| S SOCK
        Socket
type stats = {
    st_dev : int ;
        Device number
    st_ino : int ;
        Inode number
    st_kind : file_kind ;
        Kind of the file
    st_perm : file_perm ;
        Access rights
    st_nlink : int ;
        Number of links
    st_uid : int ;
        User id of the owner
    st_gid : int ;
        Group ID of the file's group
    st_rdev : int ;
        Device minor number
    st_size : int ;
        Size in bytes
    st_atime : float ;
        Last access time
    st_mtime : float ;

```

```

        Last modification time
    st_ctime : float ;
        Last status change time
}

    The information returned by the Unix.stat[49] calls.

val stat : string -> stats
    Return the information for the named file.

val lstat : string -> stats
    Same as Unix.stat[49], but in case the file is a symbolic link, return the information for the link itself.

val fstat : file_descr -> stats
    Return the information for the file associated with the given descriptor.

val isatty : file_descr -> bool
    Return true if the given file descriptor refers to a terminal or console window, false otherwise.

    File operations on large files
module LargeFile :
sig
    val lseek : Unix.file_descr -> int64 -> Unix.seek_command -> int64
    val truncate : string -> int64 -> unit
    val ftruncate : Unix.file_descr -> int64 -> unit
    type stats = {
        st_dev : int ;
            Device number
        st_ino : int ;
            Inode number
        st_kind : Unix.file_kind ;
            Kind of the file
        st_perm : Unix.file_perm ;
            Access rights
        st_nlink : int ;
            Number of links
        st_uid : int ;

```

```

        User id of the owner
    st_gid : int ;
        Group ID of the file's group
    st_rdev : int ;
        Device minor number
    st_size : int64 ;
        Size in bytes
    st_atime : float ;
        Last access time
    st_mtime : float ;
        Last modification time
    st_ctime : float ;
        Last status change time
}
val stat : string -> stats
val lstat : string -> stats
val fstat : Unix.file_descr -> stats
end

```

File operations on large files. This sub-module provides 64-bit variants of the functions `Unix.lseek`[49] (for positioning a file descriptor), `Unix.truncate`[49] and `Unix.ftruncate`[49] (for changing the size of a file), and `Unix.stat`[49], `Unix.lstat`[49] and `Unix.fstat`[49] (for obtaining information on files). These alternate functions represent positions and sizes by 64-bit integers (type `int64`) instead of regular integers (type `int`), thus allowing operating on files whose sizes are greater than `max_int`.

Operations on file names

```

val unlink : string -> unit
    Removes the named file

val rename : string -> string -> unit
    rename old new changes the name of a file from old to new.

val link : string -> string -> unit
    link source dest creates a hard link named dest to the file named source.

```

File permissions and ownership

```

type access_permission =
| R_OK

```

Read permission
 | W_OK
 Write permission
 | X_OK
 Execution permission
 | F_OK
 File exists
 Flags for the `Unix.access[49]` call.

`val chmod : string -> file_perm -> unit`
 Change the permissions of the named file.

`val fchmod : file_descr -> file_perm -> unit`
 Change the permissions of an opened file.

`val chown : string -> int -> int -> unit`
 Change the owner uid and owner gid of the named file.

`val fchown : file_descr -> int -> int -> unit`
 Change the owner uid and owner gid of an opened file.

`val umask : int -> int`
 Set the process's file mode creation mask, and return the previous mask.

`val access : string -> access_permission list -> unit`
 Check that the process has the given permissions over the named file. Raise `Unix_error` otherwise.

Operations on file descriptors

`val dup : file_descr -> file_descr`
 Return a new file descriptor referencing the same file as the given descriptor.

`val dup2 : file_descr -> file_descr -> unit`
`dup2 fd1 fd2` duplicates `fd1` to `fd2`, closing `fd2` if already opened.

`val set_nonblock : file_descr -> unit`
 Set the “non-blocking” flag on the given descriptor. When the non-blocking flag is set, reading on a descriptor on which there is temporarily no data available raises the `EAGAIN` or `EWOULDBLOCK` error instead of blocking; writing on a descriptor on which there is temporarily no room for writing also raises `EAGAIN` or `EWOULDBLOCK`.

`val clear_nonblock : file_descr -> unit`
 Clear the “non-blocking” flag on the given descriptor. See `Unix.set_nonblock[49]`.

val set_close_on_exec : file_descr -> unit

Set the “close-on-exec” flag on the given descriptor. A descriptor with the close-on-exec flag is automatically closed when the current process starts another program with one of the **exec** functions.

val clear_close_on_exec : file_descr -> unit

Clear the “close-on-exec” flag on the given descriptor. See **Unix.set_close_on_exec**[49].

Directories

val mkdir : string -> file_perm -> unit

Create a directory with the given permissions (see **Unix.umask**[49]).

val rmdir : string -> unit

Remove an empty directory.

val chdir : string -> unit

Change the process working directory.

val getcwd : unit -> string

Return the name of the current working directory.

val chroot : string -> unit

Change the process root directory.

type dir_handle

The type of descriptors over opened directories.

val opendir : string -> dir_handle

Open a descriptor on a directory

val readdir : dir_handle -> string

Return the next entry in a directory.

Raises End_of_file when the end of the directory has been reached.

val rewinddir : dir_handle -> unit

Reposition the descriptor to the beginning of the directory

val closedir : dir_handle -> unit

Close a directory descriptor.

Pipes and redirections

val pipe : unit -> file_descr * file_descr

Create a pipe. The first component of the result is opened for reading, that’s the exit to the pipe. The second component is opened for writing, that’s the entrance to the pipe.

```
val mkfifo : string -> file_perm -> unit
```

Create a named pipe with the given permissions (see `Unix.umask[49]`).

High-level process and redirection management

```
val create_process :
```

```
string ->
```

```
string array -> file_descr -> file_descr -> file_descr -> int
```

`create_process prog args new_stdin new_stdout new_stderr` forks a new process that executes the program in file `prog`, with arguments `args`. The pid of the new process is returned immediately; the new process executes concurrently with the current process. The standard input and outputs of the new process are connected to the descriptors `new_stdin`, `new_stdout` and `new_stderr`. Passing e.g. `stdout` for `new_stdout` prevents the redirection and causes the new process to have the same standard output as the current process. The executable file `prog` is searched in the path. The new process has the same environment as the current process.

```
val create_process_env :
```

```
string ->
```

```
string array ->
```

```
string array -> file_descr -> file_descr -> file_descr -> int
```

`create_process_env prog args env new_stdin new_stdout new_stderr` works as `Unix.create_process[49]`, except that the extra argument `env` specifies the environment passed to the program.

```
val open_process_in : string -> Pervasives.in_channel
```

High-level pipe and process management. This function runs the given command in parallel with the program. The standard output of the command is redirected to a pipe, which can be read via the returned input channel. The command is interpreted by the shell `/bin/sh` (cf. `system`).

```
val open_process_out : string -> Pervasives.out_channel
```

Same as `Unix.open_process_in[49]`, but redirect the standard input of the command to a pipe. Data written to the returned output channel is sent to the standard input of the command. Warning: writes on output channels are buffered, hence be careful to call `Pervasives.flush[17]` at the right times to ensure correct synchronization.

```
val open_process : string -> Pervasives.in_channel * Pervasives.out_channel
```

Same as `Unix.open_process_out[49]`, but redirects both the standard input and standard output of the command to pipes connected to the two returned channels. The input channel is connected to the output of the command, and the output channel to the input of the command.

```
val open_process_full :
```

```
string ->
```

```
string array ->
```

```
Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel
```

Similar to `Unix.open_process`[49], but the second argument specifies the environment passed to the command. The result is a triple of channels connected respectively to the standard output, standard input, and standard error of the command.

```
val close_process_in : Pervasives.in_channel -> process_status
    Close channels opened by Unix.open_process_in[49], wait for the associated command to
    terminate, and return its termination status.

val close_process_out : Pervasives.out_channel -> process_status
    Close channels opened by Unix.open_process_out[49], wait for the associated command to
    terminate, and return its termination status.

val close_process :
    Pervasives.in_channel * Pervasives.out_channel -> process_status
    Close channels opened by Unix.open_process[49], wait for the associated command to
    terminate, and return its termination status.

val close_process_full :
    Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel ->
    process_status
    Close channels opened by Unix.open_process_full[49], wait for the associated command to
    terminate, and return its termination status.
```

Symbolic links

```
val symlink : string -> string -> unit
    symlink source dest creates the file dest as a symbolic link to the file source.

val readlink : string -> string
    Read the contents of a link.
```

Polling

```
val select :
    file_descr list ->
    file_descr list ->
    file_descr list ->
    float -> file_descr list * file_descr list * file_descr list
    Wait until some input/output operations become possible on some channels. The three list
    arguments are, respectively, a set of descriptors to check for reading (first argument), for
    writing (second argument), or for exceptional conditions (third argument). The fourth
    argument is the maximal timeout, in seconds; a negative fourth argument means no timeout
    (unbounded wait). The result is composed of three sets of descriptors: those ready for
    reading (first component), ready for writing (second component), and over which an
    exceptional condition is pending (third component).
```

Locking

```
type lock_command =
    | F_ULOCK
```

Unlock a region

| F_LOCK

Lock a region for writing, and block if already locked

| F_TLOCK

Lock a region for writing, or fail if already locked

| F_TEST

Test a region for other process locks

| F_RLOCK

Lock a region for reading, and block if already locked

| F_TRLOCK

Lock a region for reading, or fail if already locked

Commands for `Unix.lockf`[49].

```
val lockf : file_descr -> lock_command -> int -> unit
```

`lockf fd cmd size` puts a lock on a region of the file opened as `fd`. The region starts at the current read/write position for `fd` (as set by `Unix.lseek`[49]), and extends `size` bytes forward if `size` is positive, `size` bytes backwards if `size` is negative, or to the end of the file if `size` is zero. A write lock prevents any other process from acquiring a read or write lock on the region. A read lock prevents any other process from acquiring a write lock on the region, but lets other processes acquire read locks on it.

The `F_LOCK` and `F_TLOCK` commands attempts to put a write lock on the specified region. The `F_RLOCK` and `F_TRLOCK` commands attempts to put a read lock on the specified region. If one or several locks put by another process prevent the current process from acquiring the lock, `F_LOCK` and `F_RLOCK` block until these locks are removed, while `F_TLOCK` and `F_TRLOCK` fail immediately with an exception. The `F_ULOCK` removes whatever locks the current process has on the specified region. Finally, the `F_TEST` command tests whether a write lock can be acquired on the specified region, without actually putting a lock. It returns immediately if successful, or fails otherwise.

Signals Note: installation of signal handlers is performed via the functions `Sys.signal`[22] and `Sys.set_signal`[22].

```
val kill : int -> int -> unit
```

`kill pid sig` sends signal number `sig` to the process with id `pid`. Under Windows, only the `Sys.sigkill` signal is emulated.

```
type sigprocmask_command =
```

```
| SIG_SETMASK
```

```
| SIG_BLOCK
```

```
| SIG_UNBLOCK
```

```
val sigprocmask : sigprocmask_command -> int list -> int list
```

`sigprocmask cmd sigs` changes the set of blocked signals. If `cmd` is `SIG_SETMASK`, blocked signals are set to those in the list `sigs`. If `cmd` is `SIG_BLOCK`, the signals in `sigs` are added to the set of blocked signals. If `cmd` is `SIG_UNBLOCK`, the signals in `sigs` are removed from the set of blocked signals. `sigprocmask` returns the set of previously blocked signals.

```
val sigpending : unit -> int list
```

Return the set of blocked signals that are currently pending.

```
val sigsuspend : int list -> unit
```

`sigsuspend sigs` atomically sets the blocked signals to `sigs` and waits for a non-ignored, non-blocked signal to be delivered. On return, the blocked signals are reset to their initial value.

```
val pause : unit -> unit
```

Wait until a non-ignored, non-blocked signal is delivered.

Time functions

```
type process_times = {  
  tms_utime : float ;
```

User time for the process

```
  tms_stime : float ;
```

System time for the process

```
  tms_cutime : float ;
```

User time for the children processes

```
  tms_cstime : float ;
```

System time for the children processes

```
}
```

The execution times (CPU times) of a process.

```
type tm = {
```

```
  tm_sec : int ;
```

Seconds 0..60

```
  tm_min : int ;
```

Minutes 0..59

```
  tm_hour : int ;
```

Hours 0..23

```
  tm_mday : int ;
```

Day of month 1..31

```
  tm_mon : int ;
```

Month of year 0..11

```

tm_year : int ;
    Year - 1900

tm_wday : int ;
    Day of week (Sunday is 0)

tm_yday : int ;
    Day of year 0..365

tm_isdst : bool ;
    Daylight time savings in effect
}

    The type representing wallclock time and calendar date.

val time : unit -> float
    Return the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

val gettimeofday : unit -> float
    Same as Unix.time[49], but with resolution better than 1 second.

val gmtime : float -> tm
    Convert a time in seconds, as returned by Unix.time[49], into a date and a time. Assumes
    UTC (Coordinated Universal Time), also known as GMT.

val localtime : float -> tm
    Convert a time in seconds, as returned by Unix.time[49], into a date and a time. Assumes
    the local time zone.

val mktime : tm -> float * tm
    Convert a date and time, specified by the tm argument, into a time in seconds, as returned
    by Unix.time[49]. The tm_isdst, tm_wday and tm_yday fields of tm are ignored. Also return
    a normalized copy of the given tm record, with the tm_wday, tm_yday, and tm_isdst fields
    recomputed from the other fields, and the other fields normalized (so that, e.g., 40 October
    is changed into 9 November). The tm argument is interpreted in the local time zone.

val alarm : int -> int
    Schedule a SIGALRM signal after the given number of seconds.

val sleep : int -> unit
    Stop execution for the given number of seconds.

val times : unit -> process_times
    Return the execution times of the process.

val utimes : string -> float -> float -> unit

```

Set the last access time (second arg) and last modification time (third arg) for a file. Times are expressed in seconds from 00:00:00 GMT, Jan. 1, 1970. A time of 0.0 is interpreted as the current time.

```
type interval_timer =
  | ITIMER_REAL
      decrements in real time, and sends the signal SIGALRM when expired.
  | ITIMER_VIRTUAL
      decrements in process virtual time, and sends SIGVTALRM when expired.
  | ITIMER_PROF
      (for profiling) decrements both when the process is running and when the system is
      running on behalf of the process; it sends SIGPROF when expired.
```

The three kinds of interval timers.

```
type interval_timer_status = {
  it_interval : float ;
      Period
  it_value : float ;
      Current value of the timer
}
```

The type describing the status of an interval timer

```
val getitimer : interval_timer -> interval_timer_status
      Return the current status of the given interval timer.
```

```
val setitimer :
  interval_timer ->
  interval_timer_status -> interval_timer_status
      setitimer t s sets the interval timer t and returns its previous status. The s argument is
      interpreted as follows: s.it_value, if nonzero, is the time to the next timer expiration;
      s.it_interval, if nonzero, specifies a value to be used in reloading it_value when the
      timer expires. Setting s.it_value to zero disables the timer. Setting s.it_interval to zero
      causes the timer to be disabled after its next expiration.
```

User id, group id

```
val getuid : unit -> int
      Return the user id of the user executing the process.
```

```
val geteuid : unit -> int
      Return the effective user id under which the process runs.
```

```
val setuid : int -> unit
```

Set the real user id and effective user id for the process.

```
val getgid : unit -> int
```

Return the group id of the user executing the process.

```
val getegid : unit -> int
```

Return the effective group id under which the process runs.

```
val setgid : int -> unit
```

Set the real group id and effective group id for the process.

```
val getgroups : unit -> int array
```

Return the list of groups to which the user executing the process belongs.

```
val setgroups : int array -> unit
```

`setgroups groups` sets the supplementary group IDs for the calling process. Appropriate privileges are required.

```
val initgroups : string -> int -> unit
```

`initgroups user group` initializes the group access list by reading the group database `/etc/group` and using all groups of which `user` is a member. The additional group `group` is also added to the list.

```
type passwd_entry = {  
  pw_name : string ;  
  pw_passwd : string ;  
  pw_uid : int ;  
  pw_gid : int ;  
  pw_gecos : string ;  
  pw_dir : string ;  
  pw_shell : string ;  
}
```

Structure of entries in the `passwd` database.

```
type group_entry = {  
  gr_name : string ;  
  gr_passwd : string ;  
  gr_gid : int ;  
  gr_mem : string array ;  
}
```

Structure of entries in the `groups` database.

```
val getlogin : unit -> string
```

Return the login name of the user executing the process.

```

val getpwnam : string -> passwd_entry
    Find an entry in passwd with the given name, or raise Not_found.

val getgrnam : string -> group_entry
    Find an entry in group with the given name, or raise Not_found.

val getpwuid : int -> passwd_entry
    Find an entry in passwd with the given user id, or raise Not_found.

val getgrgid : int -> group_entry
    Find an entry in group with the given group id, or raise Not_found.

Internet addresses
type inet_addr
    The abstract type of Internet addresses.

val inet_addr_of_string : string -> inet_addr
    Conversion from the printable representation of an Internet address to its internal
    representation. The argument string consists of 4 numbers separated by periods
    (XXX.YYY.ZZZ.TTT) for IPv4 addresses, and up to 8 numbers separated by colons for IPv6
    addresses. Raise Failure when given a string that does not match these formats.

val string_of_inet_addr : inet_addr -> string
    Return the printable representation of the given Internet address. See
    Unix.inet_addr_of_string[49] for a description of the printable representation.

val inet_addr_any : inet_addr
    A special IPv4 address, for use only with bind, representing all the Internet addresses that
    the host machine possesses.

val inet_addr_loopback : inet_addr
    A special IPv4 address representing the host machine (127.0.0.1).

val inet6_addr_any : inet_addr
    A special IPv6 address, for use only with bind, representing all the Internet addresses that
    the host machine possesses.

val inet6_addr_loopback : inet_addr
    A special IPv6 address representing the host machine (::1).

Sockets
type socket_domain =
    | PF_UNIX
        Unix domain

```

```

| PF_INET
    Internet domain (IPv4)

| PF_INET6
    Internet domain (IPv6)
    The type of socket domains. Not all platforms support IPv6 sockets (type PF_INET6).

type socket_type =
| SOCK_STREAM
    Stream socket

| SOCK_DGRAM
    Datagram socket

| SOCK_RAW
    Raw socket

| SOCK_SEQPACKET
    Sequenced packets socket
    The type of socket kinds, specifying the semantics of communications.

type sockaddr =
| ADDR_UNIX of string
| ADDR_INET of inet_addr * int
    The type of socket addresses. ADDR_UNIX name is a socket address in the Unix domain; name is a file name in the file system. ADDR_INET(addr,port) is a socket address in the Internet domain; addr is the Internet address of the machine, and port is the port number.

val socket : socket_domain -> socket_type -> int -> file_descr
    Create a new socket in the given domain, and with the given kind. The third argument is the protocol type; 0 selects the default protocol for that kind of sockets.

val domain_of_sockaddr : sockaddr -> socket_domain
    Return the socket domain adequate for the given socket address.

val socketpair :
    socket_domain ->
    socket_type -> int -> file_descr * file_descr
    Create a pair of unnamed sockets, connected together.

val accept : file_descr -> file_descr * sockaddr
    Accept connections on the given socket. The returned descriptor is a socket connected to the client; the returned address is the address of the connecting client.

val bind : file_descr -> sockaddr -> unit

```

Bind a socket to an address.

```
val connect : file_descr -> sockaddr -> unit
```

Connect a socket to an address.

```
val listen : file_descr -> int -> unit
```

Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

```
type shutdown_command =
```

```
| SHUTDOWN_RECEIVE
```

Close for receiving

```
| SHUTDOWN_SEND
```

Close for sending

```
| SHUTDOWN_ALL
```

Close both

The type of commands for `shutdown`.

```
val shutdown : file_descr -> shutdown_command -> unit
```

Shutdown a socket connection. `SHUTDOWN_SEND` as second argument causes reads on the other end of the connection to return an end-of-file condition. `SHUTDOWN_RECEIVE` causes writes on the other end of the connection to return a closed pipe condition (`SIGPIPE` signal).

```
val getsockname : file_descr -> sockaddr
```

Return the address of the given socket.

```
val getpeername : file_descr -> sockaddr
```

Return the address of the host connected to the given socket.

```
type msg_flag =
```

```
| MSG_OOB
```

```
| MSG_DONTROUTE
```

```
| MSG_PEEK
```

The flags for `Unix.recv`[49], `Unix.recvfrom`[49], `Unix.send`[49] and `Unix.sendto`[49].

```
val recv : file_descr -> bytes -> int -> int -> msg_flag list -> int
```

Receive data from a connected socket.

```
val recvfrom :
```

```
file_descr ->
```

```
bytes -> int -> int -> msg_flag list -> int * sockaddr
```

Receive data from an unconnected socket.

```
val send : file_descr -> bytes -> int -> int -> msg_flag list -> int
```

Send data over a connected socket.

```
val send_substring :  
  file_descr -> string -> int -> int -> msg_flag list -> int  
  Same as send, but take the data from a string instead of a byte sequence.
```

```
val sendto :  
  file_descr ->  
  bytes -> int -> int -> msg_flag list -> sockaddr -> int  
  Send data over an unconnected socket.
```

```
val sendto_substring :  
  file_descr ->  
  string -> int -> int -> msg_flag list -> sockaddr -> int  
  Same as sendto, but take the data from a string instead of a byte sequence.
```

Socket options

```
type socket_bool_option =  
  | SO_DEBUG  
    Record debugging information  
  | SO_BROADCAST  
    Permit sending of broadcast messages  
  | SO_REUSEADDR  
    Allow reuse of local addresses for bind  
  | SO_KEEPAIVE  
    Keep connection active  
  | SO_DONTROUTE  
    Bypass the standard routing algorithms  
  | SO_OOBINLINE  
    Leave out-of-band data in line  
  | SO_ACCEPTCONN  
    Report whether socket listening is enabled  
  | TCP_NODELAY  
    Control the Nagle algorithm for TCP sockets  
  | IPV6_ONLY  
    Forbid binding an IPv6 socket to an IPv4 address  
  The socket options that can be consulted with Unix.getsockopt[49] and modified with  
  Unix.setsockopt[49]. These options have a boolean (true/false) value.
```

```
type socket_int_option =  
  | SO_SNDBUF
```

Size of send buffer

| `SO_RCVBUF`

Size of received buffer

| `SO_ERROR`

Deprecated. Use `Unix.getsockopt_error[49]` instead.

| `SO_TYPE`

Report the socket type

| `SO_RCVLOWAT`

Minimum number of bytes to process for input operations

| `SO_SNDLOWAT`

Minimum number of bytes to process for output operations

The socket options that can be consulted with `Unix.getsockopt_int[49]` and modified with `Unix.setsockopt_int[49]`. These options have an integer value.

```
type socket_optint_option =
```

| `SO_LINGER`

Whether to linger on closed connections that have data present, and for how long (in seconds)

The socket options that can be consulted with `Unix.getsockopt_optint[49]` and modified with `Unix.setsockopt_optint[49]`. These options have a value of type `int option`, with `None` meaning “disabled”.

```
type socket_float_option =
```

| `SO_RCVTIMEO`

Timeout for input operations

| `SO_SNDTIMEO`

Timeout for output operations

The socket options that can be consulted with `Unix.getsockopt_float[49]` and modified with `Unix.setsockopt_float[49]`. These options have a floating-point value representing a time in seconds. The value 0 means infinite timeout.

```
val getsockopt : file_descr -> socket_bool_option -> bool
```

Return the current status of a boolean-valued option in the given socket.

```
val setsockopt : file_descr -> socket_bool_option -> bool -> unit
```

Set or clear a boolean-valued option in the given socket.

```
val getsockopt_int : file_descr -> socket_int_option -> int
```

Same as `Unix.getsockopt[49]` for an integer-valued socket option.

```
val setsockopt_int : file_descr -> socket_int_option -> int -> unit
```

Same as `Unix.setsockopt`[49] for an integer-valued socket option.

```
val getsockopt_optint : file_descr -> socket_optint_option -> int option
```

Same as `Unix.getsockopt`[49] for a socket option whose value is an `int option`.

```
val setsockopt_optint :  
  file_descr -> socket_optint_option -> int option -> unit
```

Same as `Unix.setsockopt`[49] for a socket option whose value is an `int option`.

```
val getsockopt_float : file_descr -> socket_float_option -> float
```

Same as `Unix.getsockopt`[49] for a socket option whose value is a floating-point number.

```
val setsockopt_float : file_descr -> socket_float_option -> float -> unit
```

Same as `Unix.setsockopt`[49] for a socket option whose value is a floating-point number.

```
val getsockopt_error : file_descr -> error option
```

Return the error condition associated with the given socket, and clear it.

High-level network connection functions

```
val open_connection :  
  sockaddr -> Pervasives.in_channel * Pervasives.out_channel
```

Connect to a server at the given address. Return a pair of buffered channels connected to the server. Remember to call `Pervasives.flush`[17] on the output channel at the right times to ensure correct synchronization.

```
val shutdown_connection : Pervasives.in_channel -> unit
```

“Shut down” a connection established with `Unix.open_connection`[49]; that is, transmit an end-of-file condition to the server reading on the other side of the connection. This does not fully close the file descriptor associated with the channel, which you must remember to free via `Pervasives.close_in`[17].

```
val establish_server :  
  (Pervasives.in_channel -> Pervasives.out_channel -> unit) ->  
  sockaddr -> unit
```

Establish a server on the given address. The function given as first argument is called for each connection with two buffered channels connected to the client. A new process is created for each connection. The function `Unix.establish_server`[49] never returns normally.

Host and protocol databases

```
type host_entry = {  
  h_name : string ;  
  h_aliases : string array ;  
  h_addrtype : socket_domain ;  
  h_addr_list : inet_addr array ;  
}
```

Structure of entries in the `hosts` database.

```
type protocol_entry = {  
  p_name : string ;  
  p_aliases : string array ;  
  p_proto : int ;  
}
```

Structure of entries in the `protocols` database.

```
type service_entry = {  
  s_name : string ;  
  s_aliases : string array ;  
  s_port : int ;  
  s_proto : string ;  
}
```

Structure of entries in the `services` database.

```
val gethostname : unit -> string
```

Return the name of the local host.

```
val gethostbyname : string -> host_entry
```

Find an entry in `hosts` with the given name, or raise `Not_found`.

```
val gethostbyaddr : inet_addr -> host_entry
```

Find an entry in `hosts` with the given address, or raise `Not_found`.

```
val getprotobyname : string -> protocol_entry
```

Find an entry in `protocols` with the given name, or raise `Not_found`.

```
val getprotobynumber : int -> protocol_entry
```

Find an entry in `protocols` with the given protocol number, or raise `Not_found`.

```
val getservbyname : string -> string -> service_entry
```

Find an entry in `services` with the given name, or raise `Not_found`.

```
val getservbyport : int -> string -> service_entry
```

Find an entry in `services` with the given service number, or raise `Not_found`.

```
type addr_info = {  
  ai_family : socket_domain ;  
    Socket domain  
  ai_socktype : socket_type ;  
    Socket type  
  ai_protocol : int ;
```

```

        Socket protocol number
ai_addr : sockaddr ;
        Address
ai_canonname : string ;
        Canonical host name
}
    Address information returned by Unix.getaddrinfo[49].

type getaddrinfo_option =
| AI_FAMILY of socket_domain
    Impose the given socket domain
| AI_SOCKTYPE of socket_type
    Impose the given socket type
| AI_PROTOCOL of int
    Impose the given protocol
| AI_NUMERICHOST
    Do not call name resolver, expect numeric IP address
| AI_CANONNAME
    Fill the ai_canonname field of the result
| AI_PASSIVE
    Set address to “any” address for use with Unix.bind[49]
Options to Unix.getaddrinfo[49].

```

```

val getaddrinfo :
  string -> string -> getaddrinfo_option list -> addr_info list
  getaddrinfo host service opts returns a list of Unix.addr_info[49] records describing
  socket parameters and addresses suitable for communicating with the given host and service.
  The empty list is returned if the host or service names are unknown, or the constraints
  expressed in opts cannot be satisfied.

  host is either a host name or the string representation of an IP address. host can be given
  as the empty string; in this case, the “any” address or the “loopback” address are used,
  depending whether opts contains AI_PASSIVE. service is either a service name or the string
  representation of a port number. service can be given as the empty string; in this case, the
  port field of the returned addresses is set to 0. opts is a possibly empty list of options that
  allows the caller to force a particular socket domain (e.g. IPv6 only or IPv4 only) or a
  particular socket type (e.g. TCP only or UDP only).

```

```

type name_info = {
  ni_hostname : string ;
    Name or IP address of host

```

```

    ni_service : string ;
}

```

Name of service or port number

Host and service information returned by `Unix.getnameinfo`[49].

```

type getnameinfo_option =

```

```

| NI_NOFQDN

```

Do not qualify local host names

```

| NI_NUMERICHOST

```

Always return host as IP address

```

| NI_NAMEREQD

```

Fail if host name cannot be determined

```

| NI_NUMERICSERV

```

Always return service as port number

```

| NI_DGRAM

```

Consider the service as UDP-based instead of the default TCP

Options to `Unix.getnameinfo`[49].

```

val getnameinfo : sockaddr -> getnameinfo_option list -> name_info

```

`getnameinfo addr opts` returns the host name and service name corresponding to the socket address `addr`. `opts` is a possibly empty list of options that governs how these names are obtained. Raise `Not_found` if an error occurs.

Terminal interface

Terminal interface

The following functions implement the POSIX standard terminal interface. They provide control over asynchronous communication ports and pseudo-terminals. Refer to the `termios` man page for a complete description.

```

type terminal_io = {

```

```

    mutable c_ignbrk : bool ;

```

Ignore the break condition.

```

    mutable c_brkint : bool ;

```

Signal interrupt on break condition.

```

    mutable c_ignpar : bool ;

```

Ignore characters with parity errors.

```

    mutable c_parmrk : bool ;

```

Mark parity errors.

```

    mutable c_inpck : bool ;

```

Enable parity check on input.

```

    mutable c_istrip : bool ;

```

Strip 8th bit on input characters.

`mutable c_inlcr : bool ;`
Map NL to CR on input.

`mutable c_igncr : bool ;`
Ignore CR on input.

`mutable c_icrnl : bool ;`
Map CR to NL on input.

`mutable c_ixon : bool ;`
Recognize XON/XOFF characters on input.

`mutable c_ixoff : bool ;`
Emit XON/XOFF chars to control input flow.

`mutable c_opost : bool ;`
Enable output processing.

`mutable c_obaud : int ;`
Output baud rate (0 means close connection).

`mutable c_ibaud : int ;`
Input baud rate.

`mutable c_csize : int ;`
Number of bits per character (5-8).

`mutable c_cstopb : int ;`
Number of stop bits (1-2).

`mutable c_cread : bool ;`
Reception is enabled.

`mutable c_parenb : bool ;`
Enable parity generation and detection.

`mutable c_parodd : bool ;`
Specify odd parity instead of even.

`mutable c_hupcl : bool ;`
Hang up on last close.

`mutable c_clocal : bool ;`
Ignore modem status lines.

`mutable c_isig : bool ;`
Generate signal on INTR, QUIT, SUSP.

`mutable c_icanon : bool ;`
Enable canonical processing (line buffering and editing)

```

mutable c_noflsh : bool ;
    Disable flush after INTR, QUIT, SUSP.

mutable c_echo : bool ;
    Echo input characters.

mutable c_echoe : bool ;
    Echo ERASE (to erase previous character).

mutable c_echok : bool ;
    Echo KILL (to erase the current line).

mutable c_echonl : bool ;
    Echo NL even if c_echo is not set.

mutable c_vintr : char ;
    Interrupt character (usually ctrl-C).

mutable c_vquit : char ;
    Quit character (usually ctrl-\\).

mutable c_verase : char ;
    Erase character (usually DEL or ctrl-H).

mutable c_vkill : char ;
    Kill line character (usually ctrl-U).

mutable c_veof : char ;
    End-of-file character (usually ctrl-D).

mutable c_veol : char ;
    Alternate end-of-line char. (usually none).

mutable c_vmin : int ;
    Minimum number of characters to read before the read request is satisfied.

mutable c_vtime : int ;
    Maximum read wait (in 0.1s units).

mutable c_vstart : char ;
    Start character (usually ctrl-Q).

mutable c_vstop : char ;
    Stop character (usually ctrl-S).
}

val tcgetattr : file_descr -> terminal_io
    Return the status of the terminal referred to by the given file descriptor.

```

```

type setattr_when =
  | TCSANOW
  | TCSADRAIN
  | TCSAFLUSH
val tcsetattr : file_descr -> setattr_when -> terminal_io -> unit
    Set the status of the terminal referred to by the given file descriptor. The second argument
    indicates when the status change takes place: immediately (TCSANOW), when all pending
    output has been transmitted (TCSADRAIN), or after flushing all input that has been received
    but not read (TCSAFLUSH). TCSADRAIN is recommended when changing the output
    parameters; TCSAFLUSH, when changing the input parameters.

val tcsendbreak : file_descr -> int -> unit
    Send a break condition on the given file descriptor. The second argument is the duration of
    the break, in 0.1s units; 0 means standard duration (0.25s).

val tcdrain : file_descr -> unit
    Waits until all output written on the given file descriptor has been transmitted.

type flush_queue =
  | TCIFLUSH
  | TCOFLUSH
  | TCIOFLUSH
val tcflush : file_descr -> flush_queue -> unit
    Discard data written on the given file descriptor but not yet transmitted, or data received
    but not yet read, depending on the second argument: TCIFLUSH flushes data received but
    not read, TCOFLUSH flushes data written but not transmitted, and TCIOFLUSH flushes both.

type flow_action =
  | TCOOFF
  | TCOON
  | TCIOFF
  | TCION
val tcflow : file_descr -> flow_action -> unit
    Suspend or restart reception or transmission of data on the given file descriptor, depending
    on the second argument: TCOOFF suspends output, TCOON restarts output, TCIOFF transmits a
    STOP character to suspend input, and TCION transmits a START character to restart input.

val setsid : unit -> int
    Put the calling process in a new session and detach it from its controlling terminal.

```

50 Module Str : Regular expressions and high-level string processing

Regular expressions

type regexp

The type of compiled regular expressions.

val regexp : string -> regexp

Compile a regular expression. The following constructs are recognized:

- `.` Matches any character except newline.
- `*` (postfix) Matches the preceding expression zero, one or several times
- `+` (postfix) Matches the preceding expression one or several times
- `?` (postfix) Matches the preceding expression once or not at all
- `[...]` Character set. Ranges are denoted with `-`, as in `[a-z]`. An initial `^`, as in `[^0-9]`, complements the set. To include a `]` character in a set, make it the first character of the set. To include a `-` character in a set, make it the first or the last character of the set.
- `^` Matches at beginning of line (either at the beginning of the matched string, or just after a newline character).
- `$` Matches at end of line (either at the end of the matched string, or just before a newline character).
- `\|` (infix) Alternative between two expressions.
- `\(...\)` Grouping and naming of the enclosed expression.
- `\1` The text matched by the first `\(...\)` expression (`\2` for the second expression, and so on up to `\9`).
- `\b` Matches word boundaries.
- `\` Quotes special characters. The special characters are `$.^.*+?[]`.

Note: the argument to **regexp** is usually a string literal. In this case, any backslash character in the regular expression must be doubled to make it past the OCaml string parser. For example, the following expression:

```
let r = Str.regexp "hello \\[A-Za-z]+\\" in
    Str.replace_first r "\\1" "hello world"
```

returns the string **"world"**.

In particular, if you want a regular expression that matches a single backslash character, you need to quote it in the argument to **regexp** (according to the last item of the list above) by adding a second backslash. Then you need to quote both backslashes (according to the syntax of string constants in OCaml) by doubling them again, so you need to write four backslash characters: **Str.regexp "\\\\"**.

val regexp_case_fold : string -> regexp

Same as **regexp**, but the compiled expression will match text in a case-insensitive way: uppercase and lowercase letters will be considered equivalent.

val quote : string -> string

`Str.quote s` returns a regexp string that matches exactly `s` and nothing else.

`val regexp_string : string -> regexp`

`Str.regexp_string s` returns a regular expression that matches exactly `s` and nothing else.

`val regexp_string_case_fold : string -> regexp`

`Str.regexp_string_case_fold` is similar to `Str.regexp_string[50]`, but the regexp matches in a case-insensitive way.

String matching and searching

`val string_match : regexp -> string -> int -> bool`

`string_match r s start` tests whether a substring of `s` that starts at position `start` matches the regular expression `r`. The first character of a string has position 0, as usual.

`val search_forward : regexp -> string -> int -> int`

`search_forward r s start` searches the string `s` for a substring matching the regular expression `r`. The search starts at position `start` and proceeds towards the end of the string. Return the position of the first character of the matched substring.

Raises `Not_found` if no substring matches.

`val search_backward : regexp -> string -> int -> int`

`search_backward r s last` searches the string `s` for a substring matching the regular expression `r`. The search first considers substrings that start at position `last` and proceeds towards the beginning of string. Return the position of the first character of the matched substring.

Raises `Not_found` if no substring matches.

`val string_partial_match : regexp -> string -> int -> bool`

Similar to `Str.string_match[50]`, but also returns true if the argument string is a prefix of a string that matches. This includes the case of a true complete match.

`val matched_string : string -> string`

`matched_string s` returns the substring of `s` that was matched by the last call to one of the following matching or searching functions:

- `Str.string_match[50]`
- `Str.search_forward[50]`
- `Str.search_backward[50]`
- `Str.string_partial_match[50]`
- `Str.global_substitute[50]`
- `Str.substitute_first[50]`

provided that none of the following functions was called inbetween:

- `Str.global_replace[50]`
- `Str.replace_first[50]`
- `Str.split[50]`
- `Str.bounded_split[50]`
- `Str.split_delim[50]`
- `Str.bounded_split_delim[50]`
- `Str.full_split[50]`
- `Str.bounded_full_split[50]`

Note: in the case of `global_substitute` and `substitute_first`, a call to `matched_string` is only valid within the `subst` argument, not after `global_substitute` or `substitute_first` returns.

The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

`val match_beginning : unit -> int`

`match_beginning()` returns the position of the first character of the substring that was matched by the last call to a matching or searching function (see `Str.matched_string[50]` for details).

`val match_end : unit -> int`

`match_end()` returns the position of the character following the last character of the substring that was matched by the last call to a matching or searching function (see `Str.matched_string[50]` for details).

`val matched_group : int -> string -> string`

`matched_group n s` returns the substring of `s` that was matched by the `n`th group `\(...\)` of the regular expression that was matched by the last call to a matching or searching function (see `Str.matched_string[50]` for details). The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

Raises `Not_found` if the `n`th group of the regular expression was not matched. This can happen with groups inside alternatives `|`, options `?` or repetitions `*`. For instance, the empty string will match `\(a\)*`, but `matched_group 1 ""` will raise `Not_found` because the first group itself was not matched.

`val group_beginning : int -> int`

`group_beginning n` returns the position of the first character of the substring that was matched by the `n`th group of the regular expression that was matched by the last call to a matching or searching function (see `Str.matched_string[50]` for details).

Raises

- `Not_found` if the `n`th group of the regular expression was not matched.
- `Invalid_argument` if there are fewer than `n` groups in the regular expression.

val `group_end` : `int -> int`

`group_end n` returns the position of the character following the last character of substring that was matched by the `n`th group of the regular expression that was matched by the last call to a matching or searching function (see `Str.matched_string[50]` for details).

Raises

- `Not_found` if the `n`th group of the regular expression was not matched.
- `Invalid_argument` if there are fewer than `n` groups in the regular expression.

Replacement

val `global_replace` : `regexp -> string -> string -> string`

`global_replace regexp templ s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by `templ`. The replacement template `templ` can contain `\1`, `\2`, etc; these sequences will be replaced by the text matched by the corresponding group in the regular expression. `\0` stands for the text matched by the whole regular expression.

val `replace_first` : `regexp -> string -> string -> string`

Same as `Str.global_replace[50]`, except that only the first substring matching the regular expression is replaced.

val `global_substitute` : `regexp -> (string -> string) -> string -> string`

`global_substitute regexp subst s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by the result of function `subst`. The function `subst` is called once for each matching substring, and receives `s` (the whole text) as argument.

val `substitute_first` : `regexp -> (string -> string) -> string -> string`

Same as `Str.global_substitute[50]`, except that only the first substring matching the regular expression is replaced.

val `replace_matched` : `string -> string -> string`

`replace_matched repl s` returns the replacement text `repl` in which `\1`, `\2`, etc. have been replaced by the text matched by the corresponding groups in the regular expression that was matched by the last call to a matching or searching function (see `Str.matched_string[50]` for details). `s` must be the same string that was passed to the matching or searching function.

Splitting

val `split` : `regexp -> string -> string list`

`split r s` splits `s` into substrings, taking as delimiters the substrings that match `r`, and returns the list of substrings. For instance, `split (regexp "[\t]+") s` splits `s` into blank-separated words. An occurrence of the delimiter at the beginning or at the end of the string is ignored.

```

val bounded_split : regexp -> string -> int -> string list
    Same as Str.split[50], but splits into at most n substrings, where n is the extra integer
    parameter.

val split_delim : regexp -> string -> string list
    Same as Str.split[50] but occurrences of the delimiter at the beginning and at the end of
    the string are recognized and returned as empty strings in the result. For instance,
    split_delim (regexp " ") " abc " returns [""; "abc"; ""], while split with the same
    arguments returns ["abc"].

val bounded_split_delim : regexp -> string -> int -> string list
    Same as Str.bounded_split[50], but occurrences of the delimiter at the beginning and at
    the end of the string are recognized and returned as empty strings in the result.

type split_result =
  | Text of string
  | Delim of string
val full_split : regexp -> string -> split_result list
    Same as Str.split_delim[50], but returns the delimiters as well as the substrings contained
    between delimiters. The former are tagged Delim in the result list; the latter are tagged
    Text. For instance, full_split (regexp "[{}]" ) "{ab}" returns [Delim "{"; Text
    "ab"; Delim "}"].

val bounded_full_split : regexp -> string -> int -> split_result list
    Same as Str.bounded_split_delim[50], but returns the delimiters as well as the substrings
    contained between delimiters. The former are tagged Delim in the result list; the latter are
    tagged Text.

    Extracting substrings
val string_before : string -> int -> string
    string_before s n returns the substring of all characters of s that precede position n
    (excluding the character at position n).

val string_after : string -> int -> string
    string_after s n returns the substring of all characters of s that follow position n
    (including the character at position n).

val first_chars : string -> int -> string
    first_chars s n returns the first n characters of s. This is the same function as
    Str.string_before[50].

val last_chars : string -> int -> string
    last_chars s n returns the last n characters of s.

```

51 Module Bigarray : Large, multi-dimensional, numerical arrays.

This module implements multi-dimensional arrays of integers and floating-point numbers, thereafter referred to as 'big arrays'. The implementation allows efficient sharing of large numerical arrays between OCaml code and C or Fortran numerical libraries.

Concerning the naming conventions, users of this module are encouraged to do `open Bigarray` in their source, then refer to array types and operations via short dot notation, e.g. `Array1.t` or `Array2.sub`.

Big arrays support all the OCaml ad-hoc polymorphic operations:

- comparisons (`=`, `<>`, `<=`, etc, as well as `Pervasives.compare`[17]);
- hashing (module `Hash`);
- and structured input-output (the functions from the `Marshal`[21] module, as well as `Pervasives.output_value` and `Pervasives.input_value`[17]).

Element kinds

Element kinds

Big arrays can contain elements of the following kinds:

- IEEE single precision (32 bits) floating-point numbers (`Bigarray.float32_elt`[51]),
- IEEE double precision (64 bits) floating-point numbers (`Bigarray.float64_elt`[51]),
- IEEE single precision (2 * 32 bits) floating-point complex numbers (`Bigarray.complex32_elt`[51]),
- IEEE double precision (2 * 64 bits) floating-point complex numbers (`Bigarray.complex64_elt`[51]),
- 8-bit integers (signed or unsigned) (`Bigarray.int8_signed_elt`[51] or `Bigarray.int8_unsigned_elt`[51]),
- 16-bit integers (signed or unsigned) (`Bigarray.int16_signed_elt`[51] or `Bigarray.int16_unsigned_elt`[51]),
- OCaml integers (signed, 31 bits on 32-bit architectures, 63 bits on 64-bit architectures) (`Bigarray.int_elt`[51]),
- 32-bit signed integer (`Bigarray.int32_elt`[51]),
- 64-bit signed integers (`Bigarray.int64_elt`[51]),
- platform-native signed integers (32 bits on 32-bit architectures, 64 bits on 64-bit architectures) (`Bigarray.nativeint_elt`[51]).

Each element kind is represented at the type level by one of the `*_elt` types defined below (defined with a single constructor instead of abstract types for technical injectivity reasons).

```
type float32_elt =  
  | Float32_elt  
type float64_elt =  
  | Float64_elt
```

```

type int8_signed_elt =
  | Int8_signed_elt
type int8_unsigned_elt =
  | Int8_unsigned_elt
type int16_signed_elt =
  | Int16_signed_elt
type int16_unsigned_elt =
  | Int16_unsigned_elt
type int32_elt =
  | Int32_elt
type int64_elt =
  | Int64_elt
type int_elt =
  | Int_elt
type nativeint_elt =
  | Nativeint_elt
type complex32_elt =
  | Complex32_elt
type complex64_elt =
  | Complex64_elt
type ('a, 'b) kind =
  | Float32 : (float, float32_elt) kind
  | Float64 : (float, float64_elt) kind
  | Int8_signed : (int, int8_signed_elt) kind
  | Int8_unsigned : (int, int8_unsigned_elt) kind
  | Int16_signed : (int, int16_signed_elt) kind
  | Int16_unsigned : (int, int16_unsigned_elt) kind
  | Int32 : (int32, int32_elt) kind
  | Int64 : (int64, int64_elt) kind
  | Int : (int, int_elt) kind
  | Nativeint : (nativeint, nativeint_elt) kind
  | Complex32 : (Complex.t, complex32_elt) kind
  | Complex64 : (Complex.t, complex64_elt) kind
  | Char : (char, int8_unsigned_elt) kind

```

To each element kind is associated an OCaml type, which is the type of OCaml values that can be stored in the big array or read back from it. This type is not necessarily the same as the type of the array elements proper: for instance, a big array whose elements are of kind `float32_elt` contains 32-bit single precision floats, but reading or writing one of its elements from OCaml uses the OCaml type `float`, which is 64-bit double precision floats.

The GADT type `('a, 'b) kind` captures this association of an OCaml type `'a` for values read or written in the big array, and of an element kind `'b` which represents the actual contents of the big array. Its constructors list all possible associations of OCaml

types with element kinds, and are re-exported below for backward-compatibility reasons.

Using a generalized algebraic datatype (GADT) here allows to write well-typed polymorphic functions whose return type depend on the argument type, such as:

```
let zero : type a b. (a, b) kind -> a = function
  | Float32 -> 0.0 | Complex32 -> Complex.zero
  | Float64 -> 0.0 | Complex64 -> Complex.zero
  | Int8_signed -> 0 | Int8_unsigned -> 0
  | Int16_signed -> 0 | Int16_unsigned -> 0
  | Int32 -> 0L | Int64 -> 0L
  | Int -> 0 | Nativeint -> 0n
  | Char -> '\000'
```

```
val float32 : (float, float32_elt) kind
```

See `Bigarray.char[51]`.

```
val float64 : (float, float64_elt) kind
```

See `Bigarray.char[51]`.

```
val complex32 : (Complex.t, complex32_elt) kind
```

See `Bigarray.char[51]`.

```
val complex64 : (Complex.t, complex64_elt) kind
```

See `Bigarray.char[51]`.

```
val int8_signed : (int, int8_signed_elt) kind
```

See `Bigarray.char[51]`.

```
val int8_unsigned : (int, int8_unsigned_elt) kind
```

See `Bigarray.char[51]`.

```
val int16_signed : (int, int16_signed_elt) kind
```

See `Bigarray.char[51]`.

```
val int16_unsigned : (int, int16_unsigned_elt) kind
```

See `Bigarray.char[51]`.

```
val int : (int, int_elt) kind
```

See `Bigarray.char[51]`.

```
val int32 : (int32, int32_elt) kind
```

See `Bigarray.char[51]`.

```
val int64 : (int64, int64_elt) kind
```

See `Bigarray.char[51]`.

```
val nativeint : (nativeint, nativeint_elt) kind
```

See `Bigarray.char[51]`.

```
val char : (char, int8_unsigned_elt) kind
```

As shown by the types of the values above, big arrays of kind `float32_elt` and `float64_elt` are accessed using the OCaml type `float`. Big arrays of complex kinds `complex32_elt`, `complex64_elt` are accessed with the OCaml type `Complex.t[11]`. Big arrays of integer kinds are accessed using the smallest OCaml integer type large enough to represent the array elements: `int` for 8- and 16-bit integer bigarrays, as well as OCaml-integer bigarrays; `int32` for 32-bit integer bigarrays; `int64` for 64-bit integer bigarrays; and `nativeint` for platform-native integer bigarrays. Finally, big arrays of kind `int8_unsigned_elt` can also be accessed as arrays of characters instead of arrays of small integers, by using the kind value `char` instead of `int8_unsigned`.

Array layouts

```
type c_layout =
```

```
| C_layout_typ
```

See `Bigarray.fortran_layout[51]`.

```
type fortran_layout =
```

```
| Fortran_layout_typ
```

To facilitate interoperability with existing C and Fortran code, this library supports two different memory layouts for big arrays, one compatible with the C conventions, the other compatible with the Fortran conventions.

In the C-style layout, array indices start at 0, and multi-dimensional arrays are laid out in row-major format. That is, for a two-dimensional array, all elements of row 0 are contiguous in memory, followed by all elements of row 1, etc. In other terms, the array elements at (x, y) and $(x, y+1)$ are adjacent in memory.

In the Fortran-style layout, array indices start at 1, and multi-dimensional arrays are laid out in column-major format. That is, for a two-dimensional array, all elements of column 0 are contiguous in memory, followed by all elements of column 1, etc. In other terms, the array elements at (x, y) and $(x+1, y)$ are adjacent in memory.

Each layout style is identified at the type level by the phantom types `Bigarray.c_layout[51]` and `Bigarray.fortran_layout[51]` respectively.

Supported layouts

The GADT type `'a layout` represents one of the two supported memory layouts: C-style or Fortran-style. Its constructors are re-exported as values below for backward-compatibility reasons.

```
type 'a layout =
```

```
| C_layout : c_layout layout
```

```
| Fortran_layout : fortran_layout layout
```

```
val c_layout : c_layout layout
```

```
val fortran_layout : fortran_layout layout
  Generic arrays (of arbitrarily many dimensions)
```

```
module Genarray :
```

```
  sig
```

```
    type ('a, 'b, 'c) t
```

The type `Genarray.t` is the type of big arrays with variable numbers of dimensions. Any number of dimensions between 1 and 16 is supported.

The three type parameters to `Genarray.t` identify the array element kind and layout, as follows:

- the first parameter, 'a, is the OCaml type for accessing array elements (`float`, `int`, `int32`, `int64`, `nativeint`);
- the second parameter, 'b, is the actual kind of array elements (`float32_elt`, `float64_elt`, `int8_signed_elt`, `int8_unsigned_elt`, etc);
- the third parameter, 'c, identifies the array layout (`c_layout` or `fortran_layout`).

For instance, `(float, float32_elt, fortran_layout) Genarray.t` is the type of generic big arrays containing 32-bit floats in Fortran layout; reads and writes in this array use the OCaml type `float`.

```
val create :
```

```
  ('a, 'b) Bigarray.kind ->
```

```
  'c Bigarray.layout -> int array -> ('a, 'b, 'c) t
```

`Genarray.create kind layout dimensions` returns a new big array whose element kind is determined by the parameter `kind` (one of `float32`, `float64`, `int8_signed`, etc) and whose layout is determined by the parameter `layout` (one of `c_layout` or `fortran_layout`). The `dimensions` parameter is an array of integers that indicate the size of the big array in each dimension. The length of `dimensions` determines the number of dimensions of the bigarray.

For instance, `Genarray.create int32 c_layout [|4;6;8|]` returns a fresh big array of 32-bit integers, in C layout, having three dimensions, the three dimensions being 4, 6 and 8 respectively.

Big arrays returned by `Genarray.create` are not initialized: the initial values of array elements is unspecified.

`Genarray.create` raises `Invalid_argument` if the number of dimensions is not in the range 1 to 16 inclusive, or if one of the dimensions is negative.

```
val num_dims : ('a, 'b, 'c) t -> int
```

Return the number of dimensions of the given big array.

```
val dims : ('a, 'b, 'c) t -> int array
```

`Genarray.dims a` returns all dimensions of the big array `a`, as an array of integers of length `Genarray.num_dims a`.

```
val nth_dim : ('a, 'b, 'c) t -> int -> int
```

`Genarray.nth_dim a n` returns the n -th dimension of the big array `a`. The first dimension corresponds to $n = 0$; the second dimension corresponds to $n = 1$; the last dimension, to $n = \text{Genarray.num_dims } a - 1$. Raise `Invalid_argument` if n is less than 0 or greater or equal than `Genarray.num_dims a`.

```
val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind
```

Return the kind of the given big array.

```
val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout
```

Return the layout of the given big array.

```
val get : ('a, 'b, 'c) t -> int array -> 'a
```

Read an element of a generic big array. `Genarray.get a [|i1; ...; iN|]` returns the element of `a` whose coordinates are `i1` in the first dimension, `i2` in the second dimension, ..., `iN` in the N -th dimension.

If `a` has C layout, the coordinates must be greater or equal than 0 and strictly less than the corresponding dimensions of `a`. If `a` has Fortran layout, the coordinates must be greater or equal than 1 and less or equal than the corresponding dimensions of `a`. Raise `Invalid_argument` if the array `a` does not have exactly N dimensions, or if the coordinates are outside the array bounds.

If $N > 3$, alternate syntax is provided: you can write `a.{i1, i2, ..., iN}` instead of `Genarray.get a [|i1; ...; iN|]`. (The syntax `a.{...}` with one, two or three coordinates is reserved for accessing one-, two- and three-dimensional arrays as described below.)

```
val set : ('a, 'b, 'c) t -> int array -> 'a -> unit
```

Assign an element of a generic big array. `Genarray.set a [|i1; ...; iN|] v` stores the value `v` in the element of `a` whose coordinates are `i1` in the first dimension, `i2` in the second dimension, ..., `iN` in the N -th dimension.

The array `a` must have exactly N dimensions, and all coordinates must lie inside the array bounds, as described for `Genarray.get`; otherwise, `Invalid_argument` is raised.

If $N > 3$, alternate syntax is provided: you can write `a.{i1, i2, ..., iN} <- v` instead of `Genarray.set a [|i1; ...; iN|] v`. (The syntax `a.{...} <- v` with one, two or three coordinates is reserved for updating one-, two- and three-dimensional arrays as described below.)

```
val sub_left :  
  ('a, 'b, Bigarray.c_layout) t ->  
  int -> int -> ('a, 'b, Bigarray.c_layout) t
```

Extract a sub-array of the given big array by restricting the first (left-most) dimension. `Genarray.sub_left a ofs len` returns a big array with the same number of

dimensions as **a**, and the same dimensions as **a**, except the first dimension, which corresponds to the interval `[ofs ... ofs + len - 1]` of the first dimension of **a**. No copying of elements is involved: the sub-array and the original array share the same storage space. In other terms, the element at coordinates `[|i1; ...; iN|]` of the sub-array is identical to the element at coordinates `[|i1+ofs; ...; iN|]` of the original array **a**.

Genarray.sub_left applies only to big arrays in C layout. Raise **Invalid_argument** if **ofs** and **len** do not designate a valid sub-array of **a**, that is, if **ofs** < 0, or **len** < 0, or **ofs** + **len** > **Genarray.nth_dim a 0**.

```
val sub_right :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int -> int -> ('a, 'b, Bigarray.fortran_layout) t
```

Extract a sub-array of the given big array by restricting the last (right-most) dimension. **Genarray.sub_right a ofs len** returns a big array with the same number of dimensions as **a**, and the same dimensions as **a**, except the last dimension, which corresponds to the interval `[ofs ... ofs + len - 1]` of the last dimension of **a**. No copying of elements is involved: the sub-array and the original array share the same storage space. In other terms, the element at coordinates `[|i1; ...; iN|]` of the sub-array is identical to the element at coordinates `[|i1; ...; iN+ofs|]` of the original array **a**.

Genarray.sub_right applies only to big arrays in Fortran layout. Raise **Invalid_argument** if **ofs** and **len** do not designate a valid sub-array of **a**, that is, if **ofs** < 1, or **len** < 0, or **ofs** + **len** > **Genarray.nth_dim a (Genarray.num_dims a - 1)**.

```
val slice_left :
  ('a, 'b, Bigarray.c_layout) t ->
  int array -> ('a, 'b, Bigarray.c_layout) t
```

Extract a sub-array of lower dimension from the given big array by fixing one or several of the first (left-most) coordinates. **Genarray.slice_left a [|i1; ... ; iM|]** returns the 'slice' of **a** obtained by setting the first **M** coordinates to **i1**, ..., **iM**. If **a** has **N** dimensions, the slice has dimension **N - M**, and the element at coordinates `[|j1; ...; j(N-M)|]` in the slice is identical to the element at coordinates `[|i1; ...; iM; j1; ...; j(N-M)|]` in the original array **a**. No copying of elements is involved: the slice and the original array share the same storage space.

Genarray.slice_left applies only to big arrays in C layout. Raise **Invalid_argument** if **M** >= **N**, or if `[|i1; ... ; iM|]` is outside the bounds of **a**.

```
val slice_right :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int array -> ('a, 'b, Bigarray.fortran_layout) t
```

Extract a sub-array of lower dimension from the given big array by fixing one or several of the last (right-most) coordinates. **Genarray.slice_right a [|i1; ... ; iM|]** returns the 'slice' of **a** obtained by setting the last **M** coordinates to **i1**, ..., **iM**. If **a** has

N dimensions, the slice has dimension $N - M$, and the element at coordinates $[|j1; \dots; j(N-M)|]$ in the slice is identical to the element at coordinates $[|j1; \dots; j(N-M); i1; \dots; iM|]$ in the original array **a**. No copying of elements is involved: the slice and the original array share the same storage space.

Genarray.slice_right applies only to big arrays in Fortran layout. Raise **Invalid_argument** if $M \geq N$, or if $[|i1; \dots; iM|]$ is outside the bounds of **a**.

```
val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit
```

Copy all elements of a big array in another big array. **Genarray.blit src dst** copies all elements of **src** into **dst**. Both arrays **src** and **dst** must have the same number of dimensions and equal dimensions. Copying a sub-array of **src** to a sub-array of **dst** can be achieved by applying **Genarray.blit** to sub-array or slices of **src** and **dst**.

```
val fill : ('a, 'b, 'c) t -> 'a -> unit
```

Set all elements of a big array to a given value. **Genarray.fill a v** stores the value **v** in all elements of the big array **a**. Setting only some elements of **a** to **v** can be achieved by applying **Genarray.fill** to a sub-array or a slice of **a**.

```
val map_file :
  Unix.file_descr ->
  ?pos:int64 ->
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> bool -> int array -> ('a, 'b, 'c) t
```

Memory mapping of a file as a big array. **Genarray.map_file fd kind layout shared dims** returns a big array of kind **kind**, layout **layout**, and dimensions as specified in **dims**. The data contained in this big array are the contents of the file referred to by the file descriptor **fd** (as opened previously with **Unix.openfile**, for example). The optional **pos** parameter is the byte offset in the file of the data being mapped; it defaults to 0 (map from the beginning of the file).

If **shared** is **true**, all modifications performed on the array are reflected in the file. This requires that **fd** be opened with write permissions. If **shared** is **false**, modifications performed on the array are done in memory only, using copy-on-write of the modified pages; the underlying file is not affected.

Genarray.map_file is much more efficient than reading the whole file in a big array, modifying that big array, and writing it afterwards.

To adjust automatically the dimensions of the big array to the actual size of the file, the major dimension (that is, the first dimension for an array with C layout, and the last dimension for an array with Fortran layout) can be given as **-1**.

Genarray.map_file then determines the major dimension from the size of the file. The file must contain an integral number of sub-arrays as determined by the non-major dimensions, otherwise **Failure** is raised.

If all dimensions of the big array are given, the file size is matched against the size of the big array. If the file is larger than the big array, only the initial portion of the file is

mapped to the big array. If the file is smaller than the big array, the file is automatically grown to the size of the big array. This requires write permissions on `fd`. Array accesses are bounds-checked, but the bounds are determined by the initial call to `map_file`. Therefore, you should make sure no other process modifies the mapped file while you're accessing it, or a SIGBUS signal may be raised. This happens, for instance, if the file is shrunk.

This function raises `Sys_error` in the case of any errors from the underlying system calls. `Invalid_argument` or `Failure` may be raised in cases where argument validation fails.

end

One-dimensional arrays

module Array1 :

sig

type ('a, 'b, 'c) t

The type of one-dimensional big arrays whose elements have OCaml type 'a, representation kind 'b, and memory layout 'c.

val create :

('a, 'b) Bigarray.kind ->

'c Bigarray.layout -> int -> ('a, 'b, 'c) t

`Array1.create kind layout dim` returns a new bigarray of one dimension, whose size is `dim`. `kind` and `layout` determine the array element kind and the array layout as described for `Genarray.create`.

val dim : ('a, 'b, 'c) t -> int

Return the size (dimension) of the given one-dimensional big array.

val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind

Return the kind of the given big array.

val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout

Return the layout of the given big array.

val get : ('a, 'b, 'c) t -> int -> 'a

`Array1.get a x`, or alternatively `a.{x}`, returns the element of `a` at index `x`. `x` must be greater or equal than 0 and strictly less than `Array1.dim a` if `a` has C layout. If `a` has Fortran layout, `x` must be greater or equal than 1 and less or equal than `Array1.dim a`. Otherwise, `Invalid_argument` is raised.

val set : ('a, 'b, 'c) t -> int -> 'a -> unit

`Array1.set a x v`, also written `a.{x} <- v`, stores the value `v` at index `x` in `a`. `x` must be inside the bounds of `a` as described in `Bigarray.Array1.get[51]`; otherwise, `Invalid_argument` is raised.

```
val sub : ('a, 'b, 'c) t ->
  int -> int -> ('a, 'b, 'c) t
```

Extract a sub-array of the given one-dimensional big array. See `Genarray.sub_left` for more details.

```
val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit
```

Copy the first big array to the second big array. See `Genarray.blit` for more details.

```
val fill : ('a, 'b, 'c) t -> 'a -> unit
```

Fill the given big array with the given value. See `Genarray.fill` for more details.

```
val of_array :
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> 'a array -> ('a, 'b, 'c) t
```

Build a one-dimensional big array initialized from the given array.

```
val map_file :
  Unix.file_descr ->
  ?pos:int64 ->
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> bool -> int -> ('a, 'b, 'c) t
```

Memory mapping of a file as a one-dimensional big array. See `Bigarray.Genarray.map_file[51]` for more details.

```
val unsafe_get : ('a, 'b, 'c) t -> int -> 'a
```

Like `Bigarray.Array1.get[51]`, but bounds checking is not always performed. Use with caution and only when the program logic guarantees that the access is within bounds.

```
val unsafe_set : ('a, 'b, 'c) t -> int -> 'a -> unit
```

Like `Bigarray.Array1.set[51]`, but bounds checking is not always performed. Use with caution and only when the program logic guarantees that the access is within bounds.

end

One-dimensional arrays. The `Array1` structure provides operations similar to those of `Bigarray.Genarray[51]`, but specialized to the case of one-dimensional arrays. (The `Array2` and `Array3` structures below provide operations specialized for two- and three-dimensional arrays.) Statically knowing the number of dimensions of the array allows faster operations, and more precise static type-checking.

Two-dimensional arrays

```
module Array2 :
```

```
sig
```

```
  type ('a, 'b, 'c) t
```

The type of two-dimensional big arrays whose elements have OCaml type 'a, representation kind 'b, and memory layout 'c.

```
  val create :
```

```
    ('a, 'b) Bigarray.kind ->
```

```
    'c Bigarray.layout -> int -> int -> ('a, 'b, 'c) t
```

`Array2.create kind layout dim1 dim2` returns a new bigarray of two dimension, whose size is `dim1` in the first dimension and `dim2` in the second dimension. `kind` and `layout` determine the array element kind and the array layout as described for `Bigarray.Genarray.create`[51].

```
  val dim1 : ('a, 'b, 'c) t -> int
```

Return the first dimension of the given two-dimensional big array.

```
  val dim2 : ('a, 'b, 'c) t -> int
```

Return the second dimension of the given two-dimensional big array.

```
  val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind
```

Return the kind of the given big array.

```
  val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout
```

Return the layout of the given big array.

```
  val get : ('a, 'b, 'c) t -> int -> int -> 'a
```

`Array2.get a x y`, also written `a.{x,y}`, returns the element of `a` at coordinates (`x`, `y`). `x` and `y` must be within the bounds of `a`, as described for `Bigarray.Genarray.get`[51]; otherwise, `Invalid_argument` is raised.

```
  val set : ('a, 'b, 'c) t -> int -> int -> 'a -> unit
```

`Array2.set a x y v`, or alternatively `a.{x,y} <- v`, stores the value `v` at coordinates (`x`, `y`) in `a`. `x` and `y` must be within the bounds of `a`, as described for `Bigarray.Genarray.set`[51]; otherwise, `Invalid_argument` is raised.

```
  val sub_left :
```

```
    ('a, 'b, Bigarray.c_layout) t ->
```

```
    int -> int -> ('a, 'b, Bigarray.c_layout) t
```

Extract a two-dimensional sub-array of the given two-dimensional big array by restricting the first dimension. See `Bigarray.Genarray.sub_left[51]` for more details. `Array2.sub_left` applies only to arrays with C layout.

```
val sub_right :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int -> int -> ('a, 'b, Bigarray.fortran_layout) t
```

Extract a two-dimensional sub-array of the given two-dimensional big array by restricting the second dimension. See `Bigarray.Genarray.sub_right[51]` for more details. `Array2.sub_right` applies only to arrays with Fortran layout.

```
val slice_left :
  ('a, 'b, Bigarray.c_layout) t ->
  int -> ('a, 'b, Bigarray.c_layout) Bigarray.Array1.t
```

Extract a row (one-dimensional slice) of the given two-dimensional big array. The integer parameter is the index of the row to extract. See `Bigarray.Genarray.slice_left[51]` for more details. `Array2.slice_left` applies only to arrays with C layout.

```
val slice_right :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array1.t
```

Extract a column (one-dimensional slice) of the given two-dimensional big array. The integer parameter is the index of the column to extract. See `Bigarray.Genarray.slice_right[51]` for more details. `Array2.slice_right` applies only to arrays with Fortran layout.

```
val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit
```

Copy the first big array to the second big array. See `Bigarray.Genarray.blit[51]` for more details.

```
val fill : ('a, 'b, 'c) t -> 'a -> unit
```

Fill the given big array with the given value. See `Bigarray.Genarray.fill[51]` for more details.

```
val of_array :
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> 'a array array -> ('a, 'b, 'c) t
```

Build a two-dimensional big array initialized from the given array of arrays.

```
val map_file :
  Unix.file_descr ->
  ?pos:int64 ->
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> bool -> int -> int -> ('a, 'b, 'c) t
```

Memory mapping of a file as a two-dimensional big array. See `Bigarray.Genarray.map_file`[51] for more details.

```
val unsafe_get : ('a, 'b, 'c) t -> int -> int -> 'a
```

Like `Bigarray.Array2.get`[51], but bounds checking is not always performed.

```
val unsafe_set : ('a, 'b, 'c) t -> int -> int -> 'a -> unit
```

Like `Bigarray.Array2.set`[51], but bounds checking is not always performed.

end

Two-dimensional arrays. The `Array2` structure provides operations similar to those of `Bigarray.Genarray`[51], but specialized to the case of two-dimensional arrays.

Three-dimensional arrays

```
module Array3 :
```

```
sig
```

```
type ('a, 'b, 'c) t
```

The type of three-dimensional big arrays whose elements have OCaml type `'a`, representation kind `'b`, and memory layout `'c`.

```
val create :
```

```
('a, 'b) Bigarray.kind ->
```

```
'c Bigarray.layout -> int -> int -> int -> ('a, 'b, 'c) t
```

`Array3.create kind layout dim1 dim2 dim3` returns a new bigarray of three dimension, whose size is `dim1` in the first dimension, `dim2` in the second dimension, and `dim3` in the third. `kind` and `layout` determine the array element kind and the array layout as described for `Bigarray.Genarray.create`[51].

```
val dim1 : ('a, 'b, 'c) t -> int
```

Return the first dimension of the given three-dimensional big array.

```
val dim2 : ('a, 'b, 'c) t -> int
```

Return the second dimension of the given three-dimensional big array.

```
val dim3 : ('a, 'b, 'c) t -> int
```

Return the third dimension of the given three-dimensional big array.

```
val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind
```

Return the kind of the given big array.

```
val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout
```

Return the layout of the given big array.

```
val get : ('a, 'b, 'c) t -> int -> int -> int -> 'a
```

`Array3.get a x y z`, also written `a.{x,y,z}`, returns the element of `a` at coordinates `(x, y, z)`. `x`, `y` and `z` must be within the bounds of `a`, as described for `Bigarray.Genarray.get[51]`; otherwise, `Invalid_argument` is raised.

```
val set : ('a, 'b, 'c) t -> int -> int -> int -> 'a -> unit
```

`Array3.set a x y v`, or alternatively `a.{x,y,z} <- v`, stores the value `v` at coordinates `(x, y, z)` in `a`. `x`, `y` and `z` must be within the bounds of `a`, as described for `Bigarray.Genarray.set[51]`; otherwise, `Invalid_argument` is raised.

```
val sub_left :
```

```
('a, 'b, Bigarray.c_layout) t ->
```

```
int -> int -> ('a, 'b, Bigarray.c_layout) t
```

Extract a three-dimensional sub-array of the given three-dimensional big array by restricting the first dimension. See `Bigarray.Genarray.sub_left[51]` for more details. `Array3.sub_left` applies only to arrays with C layout.

```
val sub_right :
```

```
('a, 'b, Bigarray.fortran_layout) t ->
```

```
int -> int -> ('a, 'b, Bigarray.fortran_layout) t
```

Extract a three-dimensional sub-array of the given three-dimensional big array by restricting the second dimension. See `Bigarray.Genarray.sub_right[51]` for more details. `Array3.sub_right` applies only to arrays with Fortran layout.

```
val slice_left_1 :
```

```
('a, 'b, Bigarray.c_layout) t ->
```

```
int -> int -> ('a, 'b, Bigarray.c_layout) Bigarray.Array1.t
```

Extract a one-dimensional slice of the given three-dimensional big array by fixing the first two coordinates. The integer parameters are the coordinates of the slice to extract. See `Bigarray.Genarray.slice_left[51]` for more details. `Array3.slice_left_1` applies only to arrays with C layout.

```
val slice_right_1 :
```

```
('a, 'b, Bigarray.fortran_layout) t ->
```

```
int -> int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array1.t
```

Extract a one-dimensional slice of the given three-dimensional big array by fixing the last two coordinates. The integer parameters are the coordinates of the slice to extract. See `Bigarray.Genarray.slice_right[51]` for more details. `Array3.slice_right_1` applies only to arrays with Fortran layout.

```
val slice_left_2 :
```

```
('a, 'b, Bigarray.c_layout) t ->
```

```
int -> ('a, 'b, Bigarray.c_layout) Bigarray.Array2.t
```

Extract a two-dimensional slice of the given three-dimensional big array by fixing the first coordinate. The integer parameter is the first coordinate of the slice to extract. See `Bigarray.Genarray.slice_left[51]` for more details. `Array3.slice_left_2` applies only to arrays with C layout.

```
val slice_right_2 :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array2.t
```

Extract a two-dimensional slice of the given three-dimensional big array by fixing the last coordinate. The integer parameter is the coordinate of the slice to extract. See `Bigarray.Genarray.slice_right[51]` for more details. `Array3.slice_right_2` applies only to arrays with Fortran layout.

```
val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit
```

Copy the first big array to the second big array. See `Bigarray.Genarray.blit[51]` for more details.

```
val fill : ('a, 'b, 'c) t -> 'a -> unit
```

Fill the given big array with the given value. See `Bigarray.Genarray.fill[51]` for more details.

```
val of_array :
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> 'a array array array -> ('a, 'b, 'c) t
```

Build a three-dimensional big array initialized from the given array of arrays of arrays.

```
val map_file :
  Unix.file_descr ->
  ?pos:int64 ->
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout ->
  bool -> int -> int -> int -> ('a, 'b, 'c) t
```

Memory mapping of a file as a three-dimensional big array. See `Bigarray.Genarray.map_file[51]` for more details.

```
val unsafe_get : ('a, 'b, 'c) t -> int -> int -> int -> 'a
```

Like `Bigarray.Array3.get[51]`, but bounds checking is not always performed.

```
val unsafe_set : ('a, 'b, 'c) t -> int -> int -> int -> 'a -> unit
```

Like `Bigarray.Array3.set[51]`, but bounds checking is not always performed.

end

Three-dimensional arrays. The `Array3` structure provides operations similar to those of `Bigarray.Genarray`[51], but specialized to the case of three-dimensional arrays.

Coercions between generic big arrays and fixed-dimension big arrays

```
val genarray_of_array1 : ('a, 'b, 'c) Array1.t -> ('a, 'b, 'c) Genarray.t
```

Return the generic big array corresponding to the given one-dimensional big array.

```
val genarray_of_array2 : ('a, 'b, 'c) Array2.t -> ('a, 'b, 'c) Genarray.t
```

Return the generic big array corresponding to the given two-dimensional big array.

```
val genarray_of_array3 : ('a, 'b, 'c) Array3.t -> ('a, 'b, 'c) Genarray.t
```

Return the generic big array corresponding to the given three-dimensional big array.

```
val array1_of_genarray : ('a, 'b, 'c) Genarray.t -> ('a, 'b, 'c) Array1.t
```

Return the one-dimensional big array corresponding to the given generic big array. Raise `Invalid_argument` if the generic big array does not have exactly one dimension.

```
val array2_of_genarray : ('a, 'b, 'c) Genarray.t -> ('a, 'b, 'c) Array2.t
```

Return the two-dimensional big array corresponding to the given generic big array. Raise `Invalid_argument` if the generic big array does not have exactly two dimensions.

```
val array3_of_genarray : ('a, 'b, 'c) Genarray.t -> ('a, 'b, 'c) Array3.t
```

Return the three-dimensional big array corresponding to the given generic big array. Raise `Invalid_argument` if the generic big array does not have exactly three dimensions.

Re-shaping big arrays

```
val reshape :
```

```
  ('a, 'b, 'c) Genarray.t ->
```

```
  int array -> ('a, 'b, 'c) Genarray.t
```

`reshape b [|d1;...;dN|]` converts the big array `b` to a N-dimensional array of dimensions `d1...dN`. The returned array and the original array `b` share their data and have the same layout. For instance, assuming that `b` is a one-dimensional array of dimension 12, `reshape b [|3;4|]` returns a two-dimensional array `b'` of dimensions 3 and 4. If `b` has C layout, the element `(x,y)` of `b'` corresponds to the element `x * 3 + y` of `b`. If `b` has Fortran layout, the element `(x,y)` of `b'` corresponds to the element `x + (y - 1) * 4` of `b`. The returned big array must have exactly the same number of elements as the original big array `b`. That is, the product of the dimensions of `b` must be equal to `i1 * ... * iN`. Otherwise, `Invalid_argument` is raised.

```
val reshape_1 : ('a, 'b, 'c) Genarray.t -> int -> ('a, 'b, 'c) Array1.t
```

Specialized version of `Bigarray.reshape`[51] for reshaping to one-dimensional arrays.

```
val reshape_2 :
```

```
  ('a, 'b, 'c) Genarray.t ->
```

```
  int -> int -> ('a, 'b, 'c) Array2.t
```

Specialized version of `Bigarray.reshape`[51] for reshaping to two-dimensional arrays.

```
val reshape_3 :  
  ('a, 'b, 'c) Genarray.t ->  
  int -> int -> int -> ('a, 'b, 'c) Array3.t
```

Specialized version of `Bigarray.reshape`[51] for reshaping to three-dimensional arrays.

52 Module Num : Operation on arbitrary-precision numbers.

Numbers (type `num`) are arbitrary-precision rational numbers, plus the special elements `1/0` (infinity) and `0/0` (undefined).

```
type num =  
  | Int of int  
  | Big_int of Big_int.big_int  
  | Ratio of Ratio.ratio  
  The type of numbers.  
  
  Arithmetic operations  
val (+/) : num -> num -> num  
  Same as Num.add_num[52].  
  
val add_num : num -> num -> num  
  Addition  
  
val minus_num : num -> num  
  Unary negation.  
  
val (-/) : num -> num -> num  
  Same as Num.sub_num[52].  
  
val sub_num : num -> num -> num  
  Subtraction  
  
val ( */ ) : num -> num -> num  
  Same as Num.mult_num[52].  
  
val mult_num : num -> num -> num  
  Multiplication  
  
val square_num : num -> num  
  Squaring  
  
val (//) : num -> num -> num
```

```

    Same as Num.div_num[52].

val div_num : num -> num -> num
    Division

val quo_num : num -> num -> num
    Euclidean division: quotient.

val mod_num : num -> num -> num
    Euclidean division: remainder.

val ( **/ ) : num -> num -> num
    Same as Num.power_num[52].

val power_num : num -> num -> num
    Exponentiation

val abs_num : num -> num
    Absolute value.

val succ_num : num -> num
    succ n is n+1

val pred_num : num -> num
    pred n is n-1

val incr_num : num Pervasives.ref -> unit
    incr r is r:=!r+1, where r is a reference to a number.

val decr_num : num Pervasives.ref -> unit
    decr r is r:=!r-1, where r is a reference to a number.

val is_integer_num : num -> bool
    Test if a number is an integer

    The four following functions approximate a number by an integer :

val integer_num : num -> num
    integer_num n returns the integer closest to n. In case of ties, rounds towards zero.

val floor_num : num -> num
    floor_num n returns the largest integer smaller or equal to n.

val round_num : num -> num
    round_num n returns the integer closest to n. In case of ties, rounds off zero.

```

```
val ceiling_num : num -> num
```

`ceiling_num n` returns the smallest integer bigger or equal to `n`.

```
val sign_num : num -> int
```

Return -1, 0 or 1 according to the sign of the argument.

Comparisons between numbers

```
val (=/) : num -> num -> bool
```

```
val (</) : num -> num -> bool
```

```
val (>/) : num -> num -> bool
```

```
val (<=/) : num -> num -> bool
```

```
val (>=/) : num -> num -> bool
```

```
val (<>/) : num -> num -> bool
```

```
val eq_num : num -> num -> bool
```

```
val lt_num : num -> num -> bool
```

```
val le_num : num -> num -> bool
```

```
val gt_num : num -> num -> bool
```

```
val ge_num : num -> num -> bool
```

```
val compare_num : num -> num -> int
```

Return -1, 0 or 1 if the first argument is less than, equal to, or greater than the second argument.

```
val max_num : num -> num -> num
```

Return the greater of the two arguments.

```
val min_num : num -> num -> num
```

Return the smaller of the two arguments.

Coercions with strings

```
val string_of_num : num -> string
```

Convert a number to a string, using fractional notation.

```
val approx_num_fix : int -> num -> string
```

See `Num.approx_num_exp[52]`.

```
val approx_num_exp : int -> num -> string
```

Approximate a number by a decimal. The first argument is the required precision. The second argument is the number to approximate. `Num.approx_num_fix[52]` uses decimal notation; the first argument is the number of digits after the decimal point. `approx_num_exp` uses scientific (exponential) notation; the first argument is the number of digits in the mantissa.

```
val num_of_string : string -> num
```

Convert a string to a number. Raise `Failure "num_of_string"` if the given string is not a valid representation of an integer

Coercions between numerical types

```
val int_of_num : num -> int
val num_of_int : int -> num
val nat_of_num : num -> Nat.nat
val num_of_nat : Nat.nat -> num
val num_of_big_int : Big_int.big_int -> num
val big_int_of_num : num -> Big_int.big_int
val ratio_of_num : num -> Ratio.ratio
val num_of_ratio : Ratio.ratio -> num
val float_of_num : num -> float
```