Previous Up Next

Module Pervasives

```
module Pervasives: sig .. end
```

The initially opened module.

This module provides the basic operations over the built-in types (numbers, booleans, 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 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 does not terminate.

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

Negation of `(=)`.

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

See `(>=)`.

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

See `(>=)`.

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

See `(>=)`.

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

Structural ordering functions. These functions coincide with the usual orderings over integers, characters, strings 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 does 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 `nan`. 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` and `Map.Make` functors, as well as the `List.sort` and `Array.sort` functions.

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

Return the smaller of the two arguments.

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

Return the greater of the two arguments.

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

`e1 == e2` tests for physical equality of `e1` and `e2`. On integers and characters, physical equality is identical to structural equality. On mutable structures, `e1 == e2` is true if and only if physical modification of `e1` also affects `e2`. On non-mutable structures, the behavior of `(==)` is implementation-dependent; however, it is guaranteed that `e1 == e2` implies `compare e1 e2 = 0`.

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

Negation of `(==)`.

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. `(&&)` 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. `(||)` should be used instead.

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 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 >= 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 mod y` and `abs(x mod y) <= abs(y)-1`. If `y = 0`, `x mod y` raises `Division_by_zero`. Notice that `x mod y` is negative if and only if `x < 0`.

`val abs : int -> int`

Return the absolute value of the argument.

`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 >= 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 >= 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 >= bitsize`.

Floating-point arithmetic

Caml'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 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 -> 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 cos : float -> float`

See `atan2`.

`val sin : float -> float`

See `atan2`.

`val tan : float -> float`

See `atan2`.

`val acos : float -> float`

See `atan2`.

`val asin : float -> float`

See `atan2`.

`val atan : float -> float`

See `atan2`.

`val atan2 : float -> float -> float`

The usual trigonometric functions.

`val cosh : float -> float`

See `tanh`.

`val sinh : float -> float`

See `tanh`.

`val tanh : float -> float`

The usual hyperbolic trigonometric functions.

`val ceil : float -> float`

See `floor`.

`val floor : float -> float`

Round the given float to an integer value. `floor f` returns the greatest integer value less than or equal to `f`. `ceil f` returns the least integer value greater than or equal to `f`.

`val abs_float : float -> float`

Return the absolute value of the argument.

`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 `float_of_int`.

`val float_of_int : int -> float`

Convert an integer to floating-point.

`val truncate : float -> int`

Same as `int_of_float`.

```
val int_of_float : float -> int
```
Truncate the given floating-point number to an integer. The result is unspecified if it 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 smallest positive float `x` such that `1.0 +. x <> 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 `classify_float` 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`.

```
val (^) : string -> string -> string
```
String concatenation.

Character operations

More character operations are provided in module `Char`.

```
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.

`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`.

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

List concatenation.

Input/output

`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 ouput 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_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_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 `open_out_gen` and `open_in_gen`.

```
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. Raise `Sys_error` if the file could not be opened.

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

Same as `open_out`, 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 `open_out`.

```
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. `open_out` and `open_out_bin` 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 : out_channel -> string -> int -> int -> unit`

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

`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 `input_binary_int` function. The format is compatible across all machines for a given version of Objective Caml.

`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 `input_value`. See the description of module `Marshal` for more information. `output_value` is equivalent to `Marshal.to_channel` 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, positionned at the beginning of the file. Raise `Sys_error` if the file could not be opened.

```
val open_in_bin : string -> in_channel
```

Same as `open_in`, 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 `open_in`.

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

`open_in 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. `open_in` and `open_in_bin` 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 -> string -> int -> int -> int
```

`input ic buf pos len` reads up to `len` characters from the given channel `ic`, storing them in string `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 `really_input` for reading exactly `len` characters.) Exception `Invalid_argument "input"` is raised if `pos` and `len` do not designate a valid substring of `buf`.

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

`really_input ic buf pos len` reads `len` characters from channel `ic`, storing them in string `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 substring of `buf`.

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

Same as `input_char`, 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 `output_binary_int`. 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 `output_value`, and return the corresponding value. This function is identical to `Marshal.from_channel`; see the description of module `Marshal` 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. Note that `close_in` may raise `Sys_error` if the operating system signals an error.

`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 .. end`

Operations on large files.

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

See modules `Printf` and `Scanf` for more operations on format strings.

```
type ('a, 'b, 'c) format = ('a, 'b, 'c, 'c) format4
```

Simplified type for format strings, included for backward compatibility with earlier releases of Objective Caml. `'a` is the type of the parameters of the format, `'c` is the result type for the "printf"-style function, and `'b` is the type of the first argument given to `%a` and `%t` printing functions.

```
val string_of_format : ('a, 'b, 'c, 'd) format4 -> string
```

Converts a format string into a string.

```
val format_of_string : ('a, 'b, 'c, 'd) format4 -> ('a, 'b, 'c, 'd) format4
```

`format_of_string s` returns a format string read from the string literal `s`.

```
val (^^) : ('a, 'b, 'c, 'd) format4 ->
       ('d, 'b, 'c, 'e) format4 -> ('a, 'b, 'c, 'e) format4
```

`f1 ^^ f2` catenates formats `f1` and `f2`. The result is a format that accepts arguments from `f1`, then arguments 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 `exit`, 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.