File: Racket File and Format Libraries

Version 7.4.0.10

September 22, 2019

Contents

Index **41**

1 Convertible: Data-Conversion Protocol

```
(require file/convertible)
```
package: base

The `file/convertible` library provides a protocol to mediate between providers of data in different possible formats and consumers of the formats. For example, a datatype that implements `prop:convertible` might be *convertible* to a GIF or PDF stream, in which case it would produce data for `'gif-bytes` or `'pdf-bytes` requests.

Any symbol can be used for a conversion request, but the following should be considered standard:

- `'text` — a string for human-readable text
- `'gif-bytes` — a byte string containing a GIF image encoding
- `'png-bytes` — a byte string containing a PNG image encoding
- `'png-bytes+bounds` — a list containing a byte string and four numbers; the byte string contains a PNG document, and the four numbers are sizing information for the image: the width, height, descent (included in the height), and extra vertical top space (included in the height), in that order
- `'png-bytes+bounds8` — a list containing a byte string and eight numbers; like `'png-bytes+bounds`, but where the image encoded that is in the byte string can be padded in each direction (to allow the drawn region to extend beyond its "bounding box"), where the extra four numbers in the list specify the amount of padding that was added to the image: left, right, top, and bottom
- `'png@2x-bytes` — like `'png-bytes`, but for an image that is intended for drawing at `1/2` scale
- `'png@2x-bytes+bounds` — like `'png-bytes+bounds`, but for an image that is intended for drawing at `1/2` scale, where the numbers in the result list are already scaled (e.g, the byte string encodes an image that is twice as wide as the first number in the resulting list)
- `'png@2x-bytes+bounds8` — like `'png-bytes+bounds8`, but but for an image that is intended for drawing at `1/2` scale, and where the numbers in the result list are already scaled
- `'svg-bytes` — a byte string containing a SVG image encoding
- `'svg-bytes+bounds` — like `'png-bytes+bounds`, but for an SVG image
- `'svg-bytes+bounds8` — like `'png-bytes+bounds8`, but for an SVG image
- `'ps-bytes` — a byte string containing a PostScript document
- `'eps-bytes` — a byte string containing an Encapsulated PostScript document

- `'eps-bytes+bounds` — like `'png-bytes+bounds`, but, but for an Encapsulated PostScript document
- `'eps-bytes+bounds8` — like `'png-bytes+bounds8`, but, but for an Encapsulated PostScript document
- `'pdf-bytes` — a byte string containing a PDF document
- `'pdf-bytes+bounds` — like `'png-bytes+bounds`, but, but for an PDF document
- `'pdf-bytes+bounds8` — like `'png-bytes+bounds8`, but, but for an PDF document

`prop:convertible`

```
: (struct-type-property/c
   (->i ([v convertible?] [request symbol?] [default default/c])
        [result
         (case request
           [(text)
            (or/c string? default/c)]
           [(gif-bytes
             png-bytes
             png@2x-bytes
             ps-bytes
             eps-bytes
             pdf-bytes
             svg-bytes)
            (or/c bytes? default/c)]
           [(png-bytes+bounds
             png@2x-bytes+bounds
             eps-bytes+bounds
             pdf-bytes+bounds)
            (or/c (list/c bytes?
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?)))
                  default/c)]
           [(png-bytes+bounds8
             png@2x-bytes+bounds8
             eps-bytes+bounds8
             pdf-bytes+bounds8)
            (or/c (list/c bytes?
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?))
                          (and/c real? (not/c negative?)))
                  default/c)]
           [else (or/c opaque-default/c any/c)])]))
```

A property whose value is invoked by `convert`.

The *`v`* argument to the procedure is the structure, the *`request`* argument is a symbol for the requested conversion, and the *`default`* argument is a value to return (typically `#f` if the conversion is not supported). The procedure's result depends on the requested conversion, as above.

The default/c contract is one generated by new-∀/c.

```
(convertible? v) → boolean?
  v : any/c
```

Returns #t if v supports the conversion protocol, #f otherwise.

```
(convert v request [default])
 → (case request
     [(text)
      (or/c string? default/c)]
     [(gif-bytes
       png-bytes
       png@2x-bytes
       ps-bytes
       eps-bytes
       pdf-bytes
       svg-bytes)
      (or/c bytes? default/c)]
     [(png-bytes+bounds
       png@2x-bytes+bounds
       eps-bytes+bounds
       pdf-bytes+bounds)
      (or/c (list/c bytes?
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?)))
            default/c)]
     [(png-bytes+bounds8
       png@2x-bytes+bounds8
       eps-bytes+bounds8
       pdf-bytes+bounds8)
      (or/c (list/c bytes?
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?))
                    (and/c real? (not/c negative?)))
            default/c)]
     [else (or/c opaque-default/c any/c)])
  v : convertible?
  request : symbol?
```

```
default : any/c = #f
```

Requests a data conversion from `v`, where *`request`* indicates the type of requested data and *`default`* is the value that the converter should return if it cannot produce data in the format indicated by *`request`*.

The `default/c` contract is one created by `new-∀/c` and it guarantees that the result of `convert` is the given default argument (or `#f` if one is not supplied).

2 gzip Compression and File Creation

```
(require file/gzip)
```
package: base

The file/gzip library provides utilities to create archive files in gzip format, or simply to compress data using the pkzip "deflate" method.

```
(gzip in-file [out-file]) → void?
  in-file : path-string?
  out-file : path-string?
           = (path-add-extension in-file ".gz" #".")
```

Compresses data to the same format as the gzip utility, writing the compressed data directly to a file. The *in-file* argument is the name of the file to compress. If the file named by *out-file* exists, it will be overwritten.

Changed in version 6.8.0.2 of package base: Changed default expression of *out-file* to use path-add-extension instead of string-append.

```
(gzip-through-ports in
                    out
                    orig-filename
                    timestamp)    → void?
  in : input-port?
  out : output-port?
  orig-filename : (or/c string? false/c)
  timestamp : exact-integer?
```

Reads the port *in* for data and compresses it to *out*, outputting the same format as the gzip utility. The *orig-filename* string is embedded in this output; *orig-filename* can be #f to omit the filename from the compressed stream. The *timestamp* number is also embedded in the output stream, as the modification date of the original file (in Unix seconds, as file-or-directory-modify-seconds would report on Unix).

```
(deflate in out) → exact-nonnegative-integer?
                   exact-nonnegative-integer?
                   exact-nonnegative-integer?
  in : input-port?
  out : output-port?
```

Writes pkzip-format "deflated" data to the port *out*, compressing data from the port *in*. The data in a file created by gzip uses this format (preceded with header information).

The result is three values: the number of bytes read from *in*, the number of bytes written to *out*, and a cyclic redundancy check (CRC) value for the input.

3 gzip Decompression

```
(require file/gunzip)
```
package: base

The file/gunzip library provides utilities to decompress archive files in gzip format, or simply to decompress data using the pkzip "inflate" method.

```
(gunzip file [output-name-filter]) → void?
  file : path-string?
  output-name-filter : (string? boolean? . -> . path-string?)
                     = (lambda (file archive-supplied?) file)
```

Extracts data that was compressed using the gzip utility (or gzip function), writing the uncompressed data directly to a file. The *file* argument is the name of the file containing compressed data. The default output file name is the original name of the compressed file as stored in *file*. If a file by this name exists, it will be overwritten. If no original name is stored in the source file, "unzipped" is used as the default output file name.

The *output-name-filter* procedure is applied to two arguments—the default destination file name and a boolean that is #t if this name was read from *file*—before the destination file is created. The return value of the file is used as the actual destination file name (to be opened with the 'truncate flag of open-output-file).

If the compressed data turns out to be corrupted, the exn:fail exception is raised.

```
(gunzip-through-ports in out) → void?
  in : input-port?
  out : output-port?
```

Reads the port *in* for compressed data that was created using the gzip utility, writing the uncompressed data to the port *out*.

If the compressed data turns out to be corrupted, the exn:fail exception is raised. The unzipping process may peek further into *in* than needed to decompress the data, but it will not consume the unneeded bytes.

```
(inflate in out) → void?
  in : input-port?
  out : output-port?
```

Reads pkzip-format "deflated" data from the port *in* and writes the uncompressed ("inflated") data to the port *out*. The data in a file created by gzip uses this format (preceded with some header information).

If the compressed data turns out to be corrupted, the exn:fail exception is raised. The inflate process may peek further into *in* than needed to decompress the data, but it will not consume the unneeded bytes.

4 `zip` File Creation

```
(require file/zip)
```
package: base

The `file/zip` library provides utilities to create `zip` archive files, which are compatible with both Windows and Unix (including Mac OS) unpacking. The actual compression is implemented by `deflate`.

```
(zip zip-file
     path ...
     [#:timestamp timestamp
      #:get-timestamp get-timestamp
      #:utc-timestamps? utc-timestamps?
      #:round-timestamps-down? round-timestamps-down?
      #:path-prefix path-prefix
      #:system-type sys-type])                          → void?
  zip-file : path-string?
  path : path-string?
  timestamp : (or/c #f exact-integer?) = #f
  get-timestamp : (path? . -> . exact-integer?)
                = (if timestamp
                      (lambda (p) timestamp)
                      file-or-directory-modify-seconds)
  utc-timestamps? : any/c = #f
  round-timestamps-down? : any/c = #f
  path-prefix : (or/c #f path-string?) = #f
  sys-type : symbol? = (system-type)
```

Creates *zip-file*, which holds the complete content of all *path*s.

The given *path*s are all expected to be relative path names of existing directories and files (i.e., relative to the current directory). If a nested path is provided as a *path*, its ancestor directories are also added to the resulting `zip` file, up to the current directory (using `pathlist-closure`).

Files are packaged as usual for `zip` files, including permission bits for both Windows and Unix (including Mac OS). The permission bits are determined by `file-or-directory-permissions`, which does not preserve the distinction between owner/group/other permissions. Also, symbolic links are always followed.

The *get-timestamp* function is used to obtain the modification date to record in the archive for a file or directory. Normally, `zip` archives record modification dates in local time, but if *utc-timestamps?* is true, then the UTC time is recorded. Timestamps in `zip` archives are precise only to two seconds; by default, the time is rounded toward the future (like WinZip or PKZIP), but time is rounded toward the past (like Java) if *round-timestamps-down?* is true.

The *sys-type* argument determines the system type recorded in the archive.

If *path-prefix* is not #f, then it prefixes the name of each path as it is written in the zip file, and directory entries are added for each element of *path-prefix*.

Changed in version 6.0.0.3 of package base: Added the #:get-timestamp and #:system-type arguments. Changed in version 6.0.1.12: Added the #:path-prefix, #:utc-timestamps?, and #:utc-timestamps-down? arguments.

```
(zip->output  paths
              [out
               #:timestamp timestamp
               #:get-timestamp get-timestamp
               #:utc-timestamps? utc-timestamps?
               #:round-timestamps-down? round-timestamps-down?
               #:path-prefix path-prefix
               #:system-type sys-type])
 → void?
  paths : (listof path-string?)
  out : output-port? = (current-output-port)
  timestamp : (or/c #f exact-integer?) = #f
  get-timestamp : (path? . -> . exact-integer?)
                = (if timestamp
                      (lambda (p) timestamp)
                      file-or-directory-modify-seconds)
  utc-timestamps? : any/c = #f
  round-timestamps-down? : any/c = #f
  path-prefix : (or/c #f path-string?) = #f
  sys-type : symbol? = (system-type)
```

Zips each of the given *paths*, and packages it as a zip "file" that is written directly to *out*. Unlike zip, the specified *paths* are included without closing over directories: if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

Changed in version 6.0.0.3 of package base: Added the #:get-timestamp and #:system-type arguments. Changed in version 6.0.1.12: Added the #:path-prefix, #:utc-timestamps?, and #:utc-timestamps-down? arguments.

```
(zip-verbose) → boolean?
(zip-verbose on?) → void?
  on? : any/c
```

A parameter that controls output during a zip operation. Setting this parameter to a true value causes zip to display to (current-error-port) the filename that is currently being compressed.

5 `zip` File Extraction

```
(require file/unzip)
```
package: base

The `file/unzip` library provides a function to extract items from a `zip` archive.

```
(unzip in
       [entry-reader
        #:preserve-timestamps? preserve-timestamps?
        #:utc-timestamps? utc-timestamps?])          → void?
  in : (or/c path-string? input-port?)
  entry-reader : (if preserve-timestamps?
                     (bytes? boolean? input-port? (or/c #f exact-integer?)
                      . -> . any)
                     (bytes? boolean? input-port? . -> . any))
               = (make-filesystem-entry-reader)
  preserve-timestamps? : any/c = #f
  utc-timestamps? : any/c = #f
```

Unzips an entire `zip` archive from *in*.

For each entry in the archive, the *entry-reader* procedure is called with three or four arguments: the byte string representing the entry name, a boolean flag indicating whether the entry represents a directory, an input port containing the inflated contents of the entry, and (if *preserve-timestamps?*) `#f` or a timestamp for a file. The default *entry-reader* unpacks entries to the filesystem; call `make-filesystem-entry-reader` to configure aspects of the unpacking, such as the destination directory.

Normally, `zip` archives record modification dates in local time, but if *utc-timestamps?* is true, then the time in the archive is interpreted as UTC.

Changed in version 6.0.0.3 of package `base`: Added the `#:preserve-timestamps?` argument.
Changed in version 6.0.1.12: Added the `#:utc-timestamps?` argument.

```
(call-with-unzip in proc) → any
  in : (or/c path-string? input-port?)
  proc : (-> path-string? any)
```

Unpacks *in* to a temporary directory, calls *proc* on the temporary directory's path, and then deletes the temporary directory while returning the result of *proc*.

Added in version 6.0.1.6 of package `base`.

```
(make-filesystem-entry-reader [#:dest dest-path
                               #:strip-count strip-count
                               #:permissive? permissive?
                               #:exists exists])
```

```
→ ((bytes? boolean? input-port?) ((or/c #f exact-integer?))
    . ->* . any)
 dest-path : (or/c path-string? #f) = #f
 strip-count : exact-nonnegative-integer? = 0
 permissive? : any/c = #f
 exists : (or/c 'skip 'error 'replace 'truncate   = 'error
                'truncate/replace 'append 'update
                'can-update 'must-truncate)
```

Creates a `zip` entry reader that can be used with either `unzip` or `unzip-entry` and whose behavior is to save entries to the local filesystem. Intermediate directories are always created if necessary before creating files. Directory entries are created as directories in the filesystem, and their entry contents are ignored.

If *dest-path* is not `#f`, every path in the archive is prefixed to determine the destination path of the extracted entry.

If *strip-count* is positive, then *strip-count* path elements are removed from the entry path from the archive (before prefixing the path with *dest-path*); if the item's path contains *strip-count* elements, then it is not extracted.

Unless *permissive?* is true, then entries with paths containing an up-directory indicator are disallowed, and a link entry whose target is an absolute path or contains an up-directory indicator is also disallowed. Absolute paths are always disallowed. A disallowed path triggers an exception.

If *exists* is `'skip` and the file for an entry already exists, then the entry is skipped. Otherwise, *exists* is passed on to `open-output-file` for writing the entry's inflated content.

Changed in version 6.0.0.3 of package `base`: Added support for the optional timestamp argument in the result function.
Changed in version 6.3: Added the `#:permissive?` argument.

```
(read-zip-directory in) → zip-directory?
  in : (or/c path-string? input-port?)
```

Reads the central directory of a `zip` file and generates a *zip directory* representing the zip file's contents. If *in* is an input port, it must support position setting via `file-position`.

This procedure performs limited I/O: it reads the list of entries from the `zip` file, but it does not inflate any of their contents.

```
(zip-directory? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a zip directory, `#f` otherwise.

```
(zip-directory-entries zipdir) → (listof bytes?)
  zipdir : zip-directory?
```

Extracts the list of entries for a `zip` archive.

```
(zip-directory-contains? zipdir name) → boolean?
  zipdir : zip-directory?
  name : (or/c bytes? path-string?)
```

Determines whether the given entry name occurs in the given zip directory. If *name* is not a byte string, it is converted using `path->zip-path`.

Directory entries match with or without trailing slashes.

```
(zip-directory-includes-directory? zipdir
                                   name)  → boolean?
  zipdir : zip-directory?
  name : (or/c bytes? path-string?)
```

Determines whether the given name is included anywhere in the given zip directory as a filesystem directory, either as an entry itself or as the containing directory of other entries. If *name* is not a byte string, it is converted using `path->zip-path`.

```
(unzip-entry  in
              zipdir
              entry
             [entry-reader
              #:preserve-timestamps? preserve-timestamps?
              #:utc-timestamps? utc-timestamps?])
 → void?
  in : (or/c path-string? input-port?)
  zipdir : zip-directory?
  entry : (or/c bytes? path-string?)
  entry-reader : (if preserve-timestamps?
                     (bytes? boolean? input-port? (or/c #f exact-integer?)
                      . -> . any)
                     (bytes? boolean? input-port? . -> . any))
                 = (make-filesystem-entry-reader)
  preserve-timestamps? : any/c = #f
  utc-timestamps? : any/c = #f
```

Unzips a single entry from a `zip` archive based on a previously read zip directory, *zipdir*, from `read-zip-directory`. If *in* is an input port, it must support position setting via `file-position`.

The *entry* parameter is a byte string whose name must be found in the zip file's central directory. If *entry* is not a byte string, it is converted using path->zip-path.

The read-entry argument is used to read the contents of the zip entry in the same way as for unzip.

If *entry* is not in *zipdir*, an exn:fail:unzip:no-such-entry exception is raised.

Changed in version 6.0.0.3 of package base: Added the #:preserve-timestamps? argument.
Changed in version 6.0.1.12: Added the #:utc-timestamps? argument.

```
(call-with-unzip-entry in entry proc) → any
  in : (or/c path-string? input-port?)
  entry : path-string?
  proc : (-> path-string? any)
```

Unpacks *entry* within *in* to a temporary directory, calls *proc* on the unpacked file's path, and then deletes the temporary directory while returning the result of *proc*.

Added in version 6.0.1.6 of package base.

```
(path->zip-path path) → bytes?
  path : path-string?
```

Converts a file name potentially containing path separators in the current platform's format to use path separators recognized by the zip file format: /.

```
(struct exn:fail:unzip:no-such-entry exn:fail (entry)
    #:extra-constructor-name make-exn:fail:unzip:no-such-entry)
  entry : bytes?
```

Raised when a requested entry cannot be found in a zip archive. The entry field is a byte string representing the requested entry name.

6 `tar` File Creation

```
(require file/tar)
```
package: base

The file/tar library provides utilities to create archive files in USTAR format, like the archive that the Unix utility pax generates. Long paths are supported using either the POSIX.1-2001/pax or GNU format for long paths. The resulting archives contain only directories, files, and symbolic links, and owner information is not preserved; the owner that is stored in the archive is always "root."

Symbolic links (on Unix and Mac OS) are not followed by default.

```
(tar  tar-file
      path ...
     [#:follow-links? follow-links?
      #:exists-ok? exists-ok?
      #:format format
      #:path-prefix path-prefix
      #:path-filter path-filter
      #:timestamp timestamp
      #:get-timestamp get-timestamp])
 → exact-nonnegative-integer?
  tar-file : path-string?
  path : path-string?
  follow-links? : any/c = #f
  exists-ok? : any/c = #f
  format : (or/c 'pax 'gnu 'ustar) = 'pax
  path-prefix : (or/c #f path-string?) = #f
  path-filter : (or/c #f (path? . -> . any/c)) = #f
  timestamp : (or/c #f exact-integer?) = #f
  get-timestamp : (path? . -> . exact-integer?)
                = (if timestamp
                      (lambda (p) timestamp)
                      file-or-directory-modify-seconds)
```

Creates *tar-file*, which holds the complete content of all *path*s. The given *path*s are all expected to be relative paths for existing directories and files (i.e., relative to the current directory). If a nested path is provided as a *path*, its ancestor directories are also added to the resulting tar file, up to the current directory (using pathlist-closure). If *follow-links?* is false, then symbolic links are included in the resulting tar file as links.

If *exists-ok?* is #f, then an exception is raised if *tar-file* exists already. If *exists-ok?* is true, then *tar-file* is truncated or replaced if it exists already.

The *format* argument determines the handling of long paths and long symbolic-link targets. If *format* is 'pax, then POSIX.1-2001/pax extensions are used. If *format* is 'gnu, then

GNU extensions are used. If *format* is 'ustar, then tar raises an error for too-long paths or symbolic-link targets.

If *path-prefix* is not #f, then it is prefixed to each path in the archive.

The *get-timestamp* function is used to obtain the modification date to record in the archive for each file or directory.

Changed in version 6.0.0.3 of package base: Added the #:get-timestamp argument.
Changed in version 6.1.1.1: Added the #:exists-ok? argument.
Changed in version 6.3.0.3: Added the #:follow-links? argument.
Changed in version 6.3.0.11: Added the #:path-filter argument.
Changed in version 6.7.0.4: Added the #:format argument and effectively changed its default from 'ustar to 'pax.
Changed in version 7.3.0.3: Added the #:timestamp argument.

```
(tar->output  paths
              [out
               #:follow-links? follow-links?
               #:format format
               #:path-prefix path-prefix
               #:path-filter path-filter
               #:timestamp timestamp
               #:get-timestamp get-timestamp])
 → exact-nonnegative-integer?
  paths : (listof path?)
  out : output-port? = (current-output-port)
  follow-links? : any/c = #f
  format : (or/c 'pax 'gnu 'ustar) = 'pax
  path-prefix : (or/c #f path-string?) = #f
  path-filter : (or/c #f (path? . -> . any/c)) = #f
  timestamp : (or/c #f exact-integer?) = #f
  get-timestamp : (path? . -> . exact-integer?)
                = (if timestamp
                      (lambda (p) timestamp)
                      file-or-directory-modify-seconds)
```

Like tar, but packages each of the given *paths* in a tar format archive that is written directly to the *out*. The specified *paths* are included as-is (except for adding *path-prefix*, if any); if a directory is specified, its content is not automatically added, and nested directories are added without parent directories.

Changed in version 6.0.0.3 of package base: Added the #:get-timestamp argument.
Changed in version 6.3.0.3: Added the #:follow-links? argument.
Changed in version 6.3.0.11: Added the #:path-filter argument.
Changed in version 6.7.0.4: Added the #:format argument and effectively changed its default from 'ustar to

'pax.

Changed in version 7.3.0.3: Added the #:timestamp argument.

```
(tar-gzip  tar-file
           paths ...
          [#:follow-links? follow-links?
           #:exists-ok? exists-ok?
           #:format format
           #:path-prefix path-prefix
           #:timestamp timestamp
           #:get-timestamp get-timestamp]) → void?
  tar-file : path-string?
  paths : path-string?
  follow-links? : any/c = #f
  exists-ok? : any/c = #f
  format : (or/c 'pax 'gnu 'ustar) = 'pax
  path-prefix : (or/c #f path-string?) = #f
  timestamp : (or/c #f exact-integer?) = #f
  get-timestamp : (path? . -> . exact-integer?)
                = (if timestamp
                      (lambda (p) timestamp)
                      file-or-directory-modify-seconds)
```

Like tar, but compresses the resulting file with gzip.

Changed in version 6.0.0.3 of package base: Added the #:get-timestamp argument.

Changed in version 6.1.1.1: Added the #:exists-ok? argument.

Changed in version 6.3.0.3: Added the #:follow-links? argument.

Changed in version 6.7.0.4: Added the #:format argument and effectively changed its default from 'ustar to 'pax.

Changed in version 7.3.0.3: Added the #:timestamp argument.

7 `tar` File Extraction

`(require file/untar)` package: `base`

The `file/untar` library provides a function to extract items from a TAR/USTAR archive using GNU and/or pax extensions to support long pathnames.

```
(untar in
       [#:dest dest-path
        #:strip-count strip-count
        #:permissive? permissive?
        #:filter filter-proc])    → void?
  in : (or/c path-string? input-port?)
  dest-path : (or/c path-string? #f) = #f
  strip-count : exact-nonnegative-integer? = 0
  permissive? : any/c = #f
  filter-proc : (path? (or/c path? #f)
                       symbol? exact-integer? (or/c path? #f)
                       exact-nonnegative-integer?
                       exact-nonnegative-integer?
                       . -> . any/c)
                = (lambda args #t)
```

Extracts TAR/USTAR content from *in*, recognizing POSIX.1-2001/pax and GNU extensions for long paths and long symbolic-link targets.

If *dest-path* is not `#f`, every path in the archive is prefixed to determine the destination path of the extracted item.

If *strip-count* is positive, then *strip-count* path elements are removed from the item path from the archive (before prefixing the path with *dest-path*); if the item's path contains *strip-count* elements, then it is not extracted.

Unless *permissive?* is true, then archive items with paths containing an up-directory indicator are disallowed, and a link item whose target is an absolute path or contains an up-directory indicator is also disallowed. Absolute paths are always disallowed. A disallowed path triggers an exception.

For each item in the archive, *filter-proc* is applied to

- the item's path as it appears in the archive;
- a destination path that is based on the path in the archive, *strip-count*, and *dest-path*–which can be `#f` if the item's path does not have *strip-count* or more elements;

- a symbol representing the item's type—`'file`, `'dir`, `'link`, `'hard-link`, `'character-special`, `'block-special`, `'fifo`, `'contiguous-file`, `'extended-header`, `'extended-header-for-next`, or `'unknown`—where only `'file`, `'dir`, or `'link` can be unpacked by `untar`;
- an exact integer representing the item's size;
- a target path for a `'link` type or `#f` for any other type;
- an integer representing the item's modification date; and
- an integer representing the item's permissions

If the result of *`filter-proc`* is `#f`, then the item is not unpacked.

Changed in version 6.3 of package `base`: Added the `#:permissive?` argument.
Changed in version 6.7.0.4: Support long paths and long symbolic-link targets using POSIX.1-2001/pax and GNU extensions.

8 tar+gzip File Extraction

```
(require file/untgz)
```
package: base

The file/untgz library provides a function to extract items from a possible gzipped TAR/USTAR archive.

```
(untgz in
       [#:dest dest-path
        #:strip-count strip-count
        #:permissive? permissive?
        #:filter filter-proc])    → void?
  in : (or/c path-string? input-port?)
  dest-path : (or/c path-string? #f) = #f
  strip-count : exact-nonnegative-integer? = 0
  permissive? : any/c = #f
  filter-proc : (path? (or/c path? #f)
                       symbol? exact-integer? (or/c path? #f)
                       exact-nonnegative-integer?
                       exact-nonnegative-integer?
                       . -> . any/c)
              = (lambda args #t)
```

The same as untar, but if *in* is in gzip form, it is gunzipped as it is unpacked.

Changed in version 6.3 of package base: Added the #:permissive? argument.

9 MD5 Message Digest

```
(require file/md5)
```
package: base

See openssl/md5 for a faster implementation with a slightly different interface.

```
(md5 in [hex-encode?]) → bytes?
  in : (or/c input-port? bytes? string?)
  hex-encode? : boolean? = #t
```

If *hex-encode?* is #t, produces a byte string containing 32 hexadecimal digits (lowercase) that is the MD5 hash of the given input stream or byte string. Otherwise produces the 16 byte long byte string that is the MD5 hash of the given input stream or byte string.

Examples:

```
> (md5 #"abc")
#"900150983cd24fb0d6963f7d28e17f72"
> (md5 #"abc" #f)
#"\220\1P\230<\322O\260\326\226?}(\341\177r"
```

10 SHA1 Message Digest

```
(require file/sha1)
```
package: base

See openssl/sha1 for a faster implementation.

```
(sha1 in [start end]) → string?
  in : (or/c bytes? input-port?)
  start : exact-nonnegative-integer? = 0
  end : (or/c #f exact-nonnegative-integer?) = #f
```

Returns a 40-character string that represents the SHA-1 hash (in hexadecimal notation) of the content from *in*. The *in*, *start*, and *end* arguments are treated the same as sha1-bytes from racket/base.

The sha1 function composes bytes->hex-string with sha1-bytes.

Example:

```
> (sha1 (open-input-bytes #"abc"))
"a9993e364706816aba3e25717850c26c9cd0d89d"
```

Changed in version 7.0.0.5 of package base: Allowed a byte string as *in* and added the *start* and *end* arguments.

```
(sha1-bytes in [start end]) → bytes?
  in : (or/c bytes? input-port?)
  start : exact-nonnegative-integer? = 0
  end : (or/c #f exact-nonnegative-integer?) = #f
```

The same as sha1-bytes from racket/base, returns a 20-byte byte string that represents the SHA-1 hash of the content from *in*.

Example:

```
> (sha1-bytes (open-input-bytes #"abc"))
#"\251\231>6G\6\201j\272>%qxP\302l\234\320\330\235"
```

Changed in version 7.0.0.5 of package base: Allowed a byte string as *in* and added the *start* and *end* arguments.

```
(bytes->hex-string bstr) → string?
  bstr : bytes?
```

Converts the given byte string to a string representation, where each byte in *bstr* is converted to its two-digit hexadecimal representation in the resulting string.

Example:

```
> (bytes->hex-string #"turtles")
"747572746c6573"
```

```
(hex-string->bytes str) → bytes?
  str : string?
```

Converts the given string to a byte string, where each pair of characters in *str* is converted to a single byte in the result.

Examples:

```
> (hex-string->bytes "70")
#"p"
> (hex-string->bytes "Af")
#"\257"
```

11 GIF File Writing

```
(require file/gif)
```
package: draw-lib

The file/gif library provides functions for writing GIF files to a stream, including GIF files with multiple images and controls (such as animated GIFs).

A GIF stream is created by gif-start, and then individual images are written with gif-add-image. Optionally, gif-add-control inserts instructions for rendering the images. The gif-end function ends the GIF stream.

A GIF stream can be in any one of the following states:

- 'init : no images or controls have been added to the stream
- 'image-or-control : another image or control can be written
- 'image : another image can be written (but not a control, since a control was written)
- 'done : nothing more can be added

```
(gif-stream? v) → boolean?
  v : any/c
```

Returns #t if v is a GIF stream created by gif-write, #f otherwise.

```
(image-ready-gif-stream? v) → boolean?
  v : any/c
```

Returns #t if v is a GIF stream that is not in 'done mode, #f otherwise.

```
(image-or-control-ready-gif-stream? v) → boolean?
  v : any/c
```

Returns #t if v is a GIF stream that is in 'init or 'image-or-control mode, #f otherwise.

```
(empty-gif-stream? v) → boolean?
  v : any/c
```

Returns #t if v is a GIF stream that in 'init mode, #f otherwise.

```
(gif-colormap? v) → boolean?
  v : any/c
```

Returns `#t` if *v* represets a colormap, `#f` otherwise. A colormap is a list whose size is a power of 2 between 2^1 and 2^8, and whose elements are vectors of size 3 containing colors (i.e., exact integers between 0 and 255 inclusive).

```
(color? v) → boolean?
  v : any/c
```

The same as `byte?`.

```
(dimension? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is an exact integer between `0` and `65535` inclusive, `#f` otherwise.

```
(gif-state stream) → symbol?
  stream : gif-stream?
```

Returns the state of *stream*.

```
(gif-start out w h bg-color cmap) → gif-stream?
  out : output-port?
  w : dimension?
  h : dimension?
  bg-color : color?
  cmap : (or/c gif-colormap? #f)
```

Writes the start of a GIF file to the given output port, and returns a GIF stream that adds to the output port.

The width and height determine a virtual space for the overall GIF image. Individual images added to the GIF stream must fit within this virtual space. The space is initialized by the given background color.

Finally, the default meaning of color numbers (such as the background color) is determined by the given colormap, but individual images within the GIF file can have their own colormaps.

A global colormap need not be supplied, in which case a colormap must be supplied for each image. Beware that *bg-color* is ill-defined if a global colormap is not provided.

```
(gif-add-image stream
               left
               top
               width
               height
               interlaced?
               cmap
               bstr)       → void?
```

```
  stream : image-ready-gif-stream?
  left : dimension?
  top : dimension?
  width : dimension?
  height : dimension?
  interlaced? : any/c
  cmap : (or/c gif-colormap? #f)
  bstr : bytes?
```

Writes an image to the given GIF stream. The *left*, *top*, *width*, and *height* values specify the location and size of the image within the overall GIF image's virtual space.

If *interlaced?* is true, then *bstr* should provide bytes ininterlaced order instead of top-to-bottom order. Interlaced order is:

- every 8th row, starting with 0
- every 8th row, starting with 4
- every 4th row, starting with 2
- every 2nd row, starting with 1

If a global color is provided with gif-start, a #f value can be provided for *cmap*.

The *bstr* argument specifies the pixel content of the image. Each byte specifies a color (i.e., an index in the colormap). Each row is provided left-to-right, and the rows provided either top-to-bottom or in interlaced order (see above). If the image is prefixed with a control that specifies an transparent index (see gif-add-control), then the corresponding "color" doesn't draw into the overall GIF image.

An exception is raised if any byte value in *bstr* is larger than the colormap's length, if the *bstr* length is not *width* times *height*, or if the *top*, *left*, *width*, and *height* dimensions specify a region beyond the overall GIF image's virtual space.

```
(gif-add-control stream
                 disposal
                 wait-for-input?
                 delay
                 transparent)    → void?
  stream : image-or-control-ready-gif-stream?
  disposal : (or/c 'any 'keep 'restore-bg 'restore-prev)
  wait-for-input? : any/c
  delay : dimension?
  transparent : (or/c color? #f)
```

Writes an image-control command to a GIF stream. Such a control must appear just before an image, and it applies to the following image.

The GIF image model involves processing images one by one, placing each image into the specified position within the overall image's virtual space. An image-control command can specify a delay before an image is added (to create animated GIFs), and it also specifies how the image should be kept or removed from the overall image before proceeding to the next one (also for GIF animation).

The *disposal* argument specifies how to proceed:

- `'any` : doesn't matter (perhaps because the next image completely overwrites the current one)
- `'keep` : leave the image in place
- `'restore-bg` : replace the image with the background color
- `'restore-prev` : restore the overall image content to the content before the image is added

If *wait-for-input?* is true, then the display program may wait for some cue from the user (perhaps a mouse click) before adding the image.

The *delay* argument specifies a delay in 1/100s of a second.

If the *transparent* argument is a color, then it determines an index that is used to represent transparent pixels in the follow image (as opposed to the color specified by the colormap for the index).

An exception is raised if a control is already added to *stream* without a corresponding image.

```
(gif-add-loop-control stream iteration) → void?
  stream : empty-gif-stream?
  iteration : dimension?
```

Writes a control command to a GIF stream for which no images or other commands have already been written. The command causes the animating sequence of images in the GIF to be repeated 'iteration-dimension' times, where 0 can be used to mean "infinity."

An exception is raise if some control or image has been added to the stream already.

```
(gif-add-comment stream bstr) → void?
  stream : image-or-control-ready-gif-stream?
  bstr : bytes?
```

Adds a generic comment to the GIF stream.

An exception is raised if an image-control command was just written to the stream (so that an image is required next).

```
(gif-end stream) → void?
  stream : image-or-control-ready-gif-stream?
```

Finishes writing a GIF file. The GIF stream's output port is not automatically closed.

An exception is raised if an image-control command was just written to the stream (so that an image is required next).

```
(quantize bstr) → bytes? gif-colormap? (or/c color? #f)
  bstr : (and/c bytes?
                (lambda (bstr)
                  (zero? (remainder (bytes-length bstr) 4))))
```

Each image in a GIF stream is limited to 256 colors, including the transparent "color," if any. The `quantize` function converts a 24-bit image (plus alpha channel) into an indexed-color image, reducing the number of colors if necessary.

Given a set of pixels expressed in ARGB format (i.e., each four bytes is a set of values for one pixel: alpha, red, blue, and green), `quantize` produces produces

- bytes for the image (i.e., a array of colors, expressed as a byte string)
- a colormap
- either `#f` or a color index for the transparent "color"

The conversion treats alpha values less than 128 as transparent pixels, and other alpha values as solid.

The quantization process uses Octrees [Gervautz1990] to construct an adaptive palette for all (non-transparent) colors in the image. This implementation is based on an article by Dean Clark [Clark1996].

To convert a collection of images all with the same quantization, simply append them for the input of a single call of `quantize`, and then break apart the result bytes.

12 ICO File Reading and Writing

```
(require file/ico)
```
package: base

The file/ico library provides functions for reading and writing ".ico" files, which contain one or more icons. Each icon is up to 256 by 256 pixels, has a particular depth (i.e., bits per pixel used to represent a color), and mask (i.e., whether a pixel is shown, except that the mask may be ignored for 32-bit icons that have an alpha value per pixel). The library also provides support for reading and writing icons in Windows executables.

```
(ico? v) → boolean?
  v : any/c
```

Returns #t if v represents an icon, #f otherwise.

```
(ico-width ico) → exact-positive-integer?
  ico : ico?
(ico-height ico) → exact-positive-integer?
  ico : ico?
(ico-depth ico) → (one-of/c 1 2 4 8 16 24 32)
  ico : ico?
```

Returns the width or height of an icon in pixels, or the depth in bits per pixel.

Changed in version 6.3 of package base: A PNG-format icon can have a width or height greater than 256.

```
(ico-format ico) → (or/c 'bmp 'png)
  ico : ico?
```

Reports the format of the icon.

Added in version 6.3 of package base.

```
(read-icos src) → (listof ico?)
  src : (or/c path-string? input-port?)
```

Parses src as an ".ico" to extract a list of icons.

```
(read-icos-from-exe src) → (listof ico?)
  src : (or/c path-string? input-port?)
```

Parses src as an ".exe" to extract the list of icons that represent the Windows executable.

```
(write-icos icos dest [#:exists exists]) → void?
  icos : (listof ico?)
```

```
  dest : (or/c path-string? output-port?)
  exists : (or/c 'error 'append 'update 'can-update = 'error
                 'replace 'truncate
                 'must-truncate 'truncate/replace)
```

Writes each icon in *icos* to *dest* as an ".ico" file. If *dest* is not an output port, *exists* is passed on to open-output-file to open *dest* for writing.

```
(replace-icos icos dest) → void?
  icos : (listof ico?)
  dest : path-string?
```

Writes icons in *icos* to replace icons in *dest* as an Windows executable. Only existing icon sizes and depths in the executable are replaced, and only when the encoding sizes match. Best matches for the existing sizes and depth are drawn from *icos* (adjusting the scale and depth of a best match as necessary).

Use replace-all-icos, instead, to replace a set of icons wholesale, especially when the set include PNG-format icons.

```
(replace-all-icos icos dest) → void?
  icos : (listof ico?)
  dest : (or/c path-string? output-port?)
```

Replaces the icon set in the executable *dest* with the given set of icons.

```
(ico->argb ico) → bytes?
  ico : ico?
```

Converts an icon in BMP format (see ico-format) to an ARGB byte string, which has the icon's pixels in left-to-right, top-to-bottom order, with four bytes (alpha, red, green, and blue channels) for each pixel.

```
(ico->png-bytes ico) → bytes?
  ico : ico?
```

Returns the bytes of a PNG encoding for an icon in PNG format (see ico-format).

Added in version 6.3 of package base.

```
(argb->ico width height bstr [#:depth depth]) → ico?
  width : (integer-in 1 256)
  height : (integer-in 1 256)
  bstr : bytes?
  depth : (one-of/c 1 2 4 8 24 32) = 32
```

Converts an ARGB byte string (in the same format as from `ico->argb`) to an icon of the given width, height, and depth in BMP format.

The *bstr* argument must have a length `(* 4 width height)`, and `(* width depth)` must be a multiple of 8.

```
(png-bytes->ico bstr) → ico?
  bstr : bytes?
```

Wraps the given PNG encoding as a PNG-encoded icon.

Added in version 6.3 of package **base**.

13 Windows Registry

```
(require file/resource)
```
package: base

```
(get-resource  section
               entry
              [value-box
               file
               #:type type])
 → (or/c #f string? bytes? exact-integer? #t)
  section : string?
  entry : string?
  value-box : (or/c #f (box/c (or/c string? bytes? exact-integer?)))
            = #f
  file : (or/c #f path-string?) = #f
  type : (or/c 'string 'bytes 'integer) = derived-from-value-box
```

Gets a value from the Windows registry or an ".ini" file. For backward compatibilty, the result is #f for platforms other than Windows. The registry is read when *file* is #f and when *section* is "HKEY_CLASSES_ROOT", "HKEY_CURRENT_CONFIG", "HKEY_CURRENT_USER", "HKEY_LOCAL_MACHINE", or "HKEY_USERS". When *file* is #f and *section* is not one of the special registry strings, then (build-path (find-system-path 'home-dir) "mred.ini") is read.

The resource value is keyed on the combination of *section* and *entry*. The result is #f if no value is found for the specified *section* and *entry*. If *value-box* is a box, then the result is #t if a value is found, and the box is filled with the value; when *value-box* is #f, the result is the found value.

The *type* argument determines how a value in the resource is converted to a Racket value. If *value-box* is a box, then the default *type* is derived from the initial box content, otherwise the default *type* is 'string.

Registry values of any format can be extracted. Values using the registry format REG_SZ are treated as strings, and values with the format REG_DWORD are treated as 32-bit signed integers. All other formats are treated as raw bytes. Data from the registry is converted to the requested *type* as follows:

- A REG_SZ registry value is converted to an integer using string->number (using 0 if the result is not an exact integer), and it is converted to bytes using string->bytes/utf-8.
- A REG_DWORD registry value is converted to a string or byte string via number->string and (for byte strings) string->bytes/utf-8.
- Any other kind of registry value is converted to a string or integer using bytes->string/utf-8 and (for integers) string->number.

Resources from `".ini"` files are always strings, and are converted like `REG_SZ` registry values.

To get the "default" value for a registry entry, use a trailing backslash. For example, the following expression gets a command line for starting a browser:

```
(get-resource "HKEY_CLASSES_ROOT"
              "htmlfile\\shell\\open\\command\\")
```

```
(write-resource  section
                 entry
                 value
                [file
                 #:type type
                 #:create-key? create-key?]) → boolean?
  section : string?
  entry : string?
  value : (or/c string? bytes? exact-integer?)
  file : (or/c path-string? #f) = #f
  type : (or/c 'string 'bytes 'integer) = 'string
  create-key? : any/c = #f
```

Write a value to the Windows registry or an `".ini"` file. For backward compatibilty, the result is `#f` for platforms other than Windows. The registry is written when *file* is `#f` and when *section* is `"HKEY_CLASSES_ROOT"`, `"HKEY_CURRENT_CONFIG"`, `"HKEY_CURRENT_USER"`, `"HKEY_LOCAL_MACHINE"`, or `"HKEY_USERS"`. When *file* is `#f` and *section* is not one of the special registry strings, then `(build-path (find-system-path 'home-dir) "mred.ini")` is written.

The resource value is keyed on the combination of *section* and *entry*. If *create-key?* is false when writing to the registry, the resource entry must already exist, otherwise the write fails. The result is `#f` if the write fails or `#t` if it succeeds.

The *type* argument determines the format of the value written to the registry: `'string` writes using the `REG_SZ` format, `'bytes` writes using the `REG_BINARY` format, and `'dword` writes using the `REG_DWORD` format. Any kind of *value* can be converted for any kind of *type* using the inverse of the conversions for `get-resource`.

When writing to an `".ini"` file, the format is always a string, independent of *type*.

14 Caching

```
(require file/cache)
```
package: base

The file/cache library provides utilities for managing a local cache of files, such as downloaded files. The cache is safe for concurrent use across processes, since it uses filesystem locks, and it isolates clients from filesystem failures.

```
(cache-file dest-file
            [#:exists-ok? exists-ok?]
             key
             cache-dir
             fetch
            [#:notify-cache-use notify-cache-use
             #:max-cache-files max-files
             #:max-cache-size max-size
             #:evict-before? evict-before?
             #:log-error-string log-error-string
             #:log-debug-string log-debug-string]) → void?
  dest-file : path-string?
  exists-ok? : any/c = #f
  key : (not/c #f)
  cache-dir : path-string?
  fetch : (-> any)
  notify-cache-use : (string? . -> . any) = void
  max-files : real? = 1024
  max-size : real? = (* 64 1024 1024)
  evict-before? : (hash? hash? . -> . boolean?)
                = (lambda (a b)
                    (< (hash-ref a 'modify-seconds)
                       (hash-ref b 'modify-seconds)))
  log-error-string : (string? . -> . any)
                   = (lambda (s) (log-error s))
  log-debug-string : (string? . -> . any)
                   = (lambda (s) (log-debug s))
```

Looks for a file in *cache-dir* previously cached with *key*, and copies it to *dest-file* (which must not exist already, unless *exists-ok?* is true) if a cached file is found. Otherwise, *fetch* is called; if *dest-file* exists after calling *fetch*, it is copied to *cache-dir* and recorded with *key*. When a cache entry is used, *notify-cache-use* is called with the name of the cache file.

When a new file is cached, *max-files* (as a file count) and *max-size* (in bytes) determine whether any previously cached files should be evicted from the cache. If so, *evict-before?* determines an order on existing cache entries for eviction; each argument to *evict-before?* is a hash table with at least the following keys:

- `'modify-seconds` — the file's modification date
- `'size` — the file's size in bytes
- `'key` — the cache entry's key
- `'name` — the cache file's name

The *log-error-string* and *log-debug-string* functions are used to record errors and debugging information.

```
(cache-remove  key
               cache-dir
              [#:log-error-string log-error-string
               #:log-debug-string log-debug-string]) → void?
  key : any/c
  cache-dir : path-string?
  log-error-string : (string? . -> . any)
                   = (lambda (s) (log-error s))
  log-debug-string : (string? . -> . any)
                   = (lambda (s) (log-debug s))
```

Removes the cache entry matching *key* (if any) from the cache in *cache-dir*, or removes all cached files if *key* is `#f`.

The *log-error-string* and *log-debug-string* functions are used to record errors and debugging information.

15 Globbing

```
(require file/glob)
```
package: base

The file/glob library implements globbing for path-string? values. A *glob* is a path string that matches a set of path strings using the following *wildcards*:

- A sextile (*) matches any sequence of characters in a file or directory name.
- Two sextiles (**) match any sequence of characters and any number of path separators.
- A question mark (?) matches any single character in a file or directory name.
- Square bracket-delimited character groups, e.g. [abc], match any character within the group. The square brackets have the same meaning in globs as in regular expressions, see §4.7.1 “Regexp Syntax”.
- If the glob ends with a path separator (/ on any (system-type), additionally \ on 'windows) then it only matches directories.

By default, wildcards will not match files or directories whose name begins with a period (aka "dotfiles"). To override, set the parameter glob-capture-dotfiles? to a non-#f value or supply a similar value using the #:capture-dotfiles? keyword.

```
glob/c : (or/c path-string? (sequence/c path-string?))
```

A flat contract that accepts a glob or a sequence of globs.

All file/glob functions accept glob/c values. These functions also recognize braces ({}) as a *meta-wildcard* for describing multiple globs.

Braces are interpreted *before* any other wildcards.

- Brace-delimited, comma-separated character groups, e.g. {foo,bar}, expand to multiple globs before the file/glob module begins matching. For example, the glob/c value "{foo,bar}.rkt" has the same meaning as '("foo.rkt" "bar.rkt").

```
(glob pattern
      [#:capture-dotfiles? capture-dotfiles?])
 → (listof path-string?)
  pattern : glob/c
  capture-dotfiles? : boolean? = (glob-capture-dotfiles?)
```

Builds a list of all paths on the current filesystem that match any glob in *pattern*. The order of paths in the result is unspecified.

If *pattern* contains the wildcard **, then glob recursively searches the filesystem to find matches. For example, the glob "/**.rkt" will search the *entire filesystem* for files or directories with a ".rkt" suffix (aka, Racket files).

Examples:

```
> (glob "*.rkt")
;; Lists all Racket files in current directory

> (glob "*/*.rkt")
;; Lists all Racket files in all sub-directories of the current directory.
;; (Does not search sub-sub-directories, etc.)

> (glob (build-path (find-system-path 'home-dir) "**" "*.rkt"))
;; Recursively searches home directory for Racket files, lists all matches.

> (glob "??.rkt")
;; Lists all Racket files in current directory with 2-
character names.

> (glob "[a-z0-9].rkt")
;; Lists all Racket files in current directory with single-
character,
;; alphanumeric names.

> (glob '("foo-bar.rkt" "foo-baz.rkt" "qux-bar.rkt" "qux-
baz.rkt"))
;; Filters the list to contain only files or directories that exist.

> (glob "{foo,qux}-{bar,baz}.rkt")
;; Same as above, returns at most 4 files.
```

```
(in-glob  pattern
          [#:capture-dotfiles? capture-dotfiles?])
 → (sequence/c path-string?)
  pattern : glob/c
  capture-dotfiles? : boolean? = (glob-capture-dotfiles?)
```

Returns a stream of all paths matching the glob *pattern*, instead of eagerly building a list.

```
(glob-match?  pattern
              path
             [#:capture-dotfiles? capture-dotfiles?]) → boolean?
  pattern : glob/c
  path : path-string?
  capture-dotfiles? : boolean? = (glob-capture-dotfiles?)
```

Analogous to `regexp-match?`; returns `#true` if *`path`* matches any glob in *`pattern`*.

`(glob-match? pattern path)` is *not* the same as:

```
(member path (glob pattern))
```

because `glob` only returns files/directories that exist, whereas `glob-match?` does not check that *`path`* exists.

This operation accesses the filesystem.

```
(glob-quote str) → string?
  str : string?
(glob-quote path) → path?
  path : path?
```

Escapes all glob wildcards and glob meta-wildcards in the given string or path string.

Examples:

```
> (glob-quote "*.rkt")
"\\*.rkt"
> (glob-quote "[Ff]ile?{zip,tar.gz}")
"\\[Ff\\]ile\\?\\{zip\\,tar.gz\\}"
> (glob-quote "]")
"\\]"
```

```
(glob-capture-dotfiles?) → boolean?
(glob-capture-dotfiles? capture-dotfiles?) → void?
  capture-dotfiles? : boolean?
  = #f
```

Determines whether wildcards match names that begin with a `#\.` character. If `#t`, the wildcards will match dotfiles. If `#f`, use a glob such as `".*"` to match dotfiles explicitly.

Bibliography

[Gervautz1990] M. Gervautz and W. Purgathofer, "A simple method for color quantization: Octree quantization," Graphics Gems, 1990.

[Clark1996] Dean Clark, "Color Quantization using Octrees," Dr. Dobbs Journal, January 1, 1996. http://www.ddj.com/184409805

Index